و ا/خریداری
۱۲۲۰۰۰/ریال
۷۶,۸,۳

An Introduction to Parallel Algorithms

Joseph JáJá
UNIVERSITY OF MARYLAND

کتابخانه مرکزی
دانشگاه صنعتی شریف
شماره ثبت امانی
113602

ADDISON-WESLEY PUBLISHING COMPANY

Reading, Massachusetts • Menlo Park, California • New York
Don Mills, Ontario • Wokingham, England • Amsterdam • Bonn
Sydney • Singapore • Tokyo • Madrid • San Juan • Milan • Paris

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial caps or all caps.

The programs and applications presented in this book have been included for their instructional value. They have been tested with care, but are not guaranteed for any particular purpose. The publisher does not offer any warranties or representations, nor does it accept any liabilities with respect to the programs or applications.

Library of Congress Cataloging-in-Publication Data

JáJá, Joseph.
An Introduction to parallel algorithms / Joseph JáJá.
p. cm.
Includes bibliographical references and index.
ISBN 0-201-54856-9
1. Parallel processing (Electronic computers) 2. Computer algorithms. I. Title.
QA76.58.J35 1992
004'.35—dc20 91-28992
CIP

QA
76
.58
.J35
1992

Reprinted with corrections November, 1992

Copyright © 1992 by Addison-Wesley Publishing Company, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America.

4 5 6 7 8 9 10 MA 9695

PREFACE

This book is an introduction to the design and analysis of parallel algorithms. There is sufficient material for a one-semester course at the senior or first-year graduate level, and for a follow-up graduate-level course covering more advanced material. Our principal model for algorithmic design is the shared-memory model; however, all of our algorithms are described at a high level similar to that provided by a data-parallel programming environment. It is assumed that the reader has some understanding of elementary discrete mathematics, basic data structures, and algorithms, at the level covered in typical undergraduate curricula in computer science and computer engineering.

The Need

Fundamental physical limitations on processing speeds will eventually force high-performance computations to be targeted principally at the exploitation of parallelism. Just as the fastest cycle times are approaching these fundamental barriers, new generations of parallel machines are emerging. It is just a matter of time until the transition affects the whole activity surrounding general-purpose computing. We hope that this book will serve as a vehicle to prepare the students in computer science and computer engineering for such a transition.

The Model

A universally accepted model for designing and analyzing sequential algorithms consists of a central processing unit with a random access memory attached to it. The success of this model is primarily due to its simplicity and to its ability to capture in a significant way the performance of sequential algorithms on von Neumann-type computers. Unfortunately, parallel computation suffers from the lack of a commonly accepted model due to the additional complexity introduced by the presence of a set of interconnected processors. In particular, the performance of a parallel algorithm seems to depend on several factors such as overall resource requirements, computational concurrency, processor allocation and scheduling, communication, and synchronization.

Our formal model for algorithmic design in this book is the *shared-memory model* or, more precisely, the *parallel random access machine* (PRAM). This formal model allows us to establish optimal results and to relate parallel complexity to complexity measures defined on classical models such as the circuit model and Turing machines. However, all our algorithms are described at a higher level called the *work-time presentation framework*. In this framework, an algorithm is described in terms of a sequence of time units, where any number of concurrent primitive operations can take place during each time unit. The performance of the parallel algorithm is measured in terms of two parameters: *work*, which is defined to be the total number of operations used by the algorithm, and *parallel time*, which is the number of time units required to execute the algorithm. This framework provides a simple machine-independent model for writing parallel algorithms. As a matter of fact, it is closely related to data-parallel algorithms that have been shown to be efficiently implementable on a wide variety of parallel machines (SIMD or MIMD, shared memory or distributed memory). The data-parallel programming model has been steadily gaining acceptance in the parallel processing community.

The Use of this Book

There are a number of ways to organize the material of this book into a one-semester course or a sequence of two one-semester courses. The first three chapters (except the sections marked with stars) and Section 4.1 represent the core material that should be covered in a first course in parallel algorithms. Chapters 5 through 10 are mostly independent of each other and

can be covered in almost any order (however, a parallel approach is not recommended!). One possible sequence for a basic course in parallel algorithms consists of Chapters 1, 2, Sections 3.1–3.3, 4.1–4.2, 4.3.1, 4.4, 5.1–5.2, 5.5, 6.1–6.2, and 8.1–8.3. Chapters 6 through 10 can be used as the basis for a second, more advanced course in parallel algorithms.

A section marked with a star represents either optional material or advanced material that can be skipped without loss of continuity. Each chapter ends with a number of exercises, some of which are straightforward applications of the ideas developed in the chapter, and others present nontrivial extensions. Exercises judged to be on the difficult side are starred. More details and extensions can be found in the references cited at the end of each chapter.

At the end of each chapter, I have cited references that are related to the material covered in the chapter. My apologies to the many authors whose work on parallel algorithms was not cited. I will consider any comments or criticism in this regard and will make appropriate updates in future editions of the book.

Comments or corrections can be communicated to the author through the e-mail address: joseph@src.umd.edu

Acknowledgments

I would like to thank the many people who have contributed significantly to the quality of this book.

The work on this book could not have been started if it were not for the generous support and the professional environment provided by the Parallel Processing Group at the National Institute of Standards and Technology (NIST), where I spent my sabbatical during the year 1989–1990. In particular, I wish to thank R. Carpenter and A. Mink for their hospitality and their help.

My colleagues at the University of Maryland have been extremely helpful in providing suggestions and constructive criticism during the preparation of the manuscript. In particular, U. Vishkin has provided me with valuable insights and comments, especially during the initial stage of this work. O. Berkman, L. Davis, B. Gasarch, R. Greenberg, S. Khuller, Y. Matias, K. Nakjima, S. Rangarajan, and R. Thurimella read portions of the manuscript and gave me corrections and suggestions for improving the quality of the book. I would also like to thank graduate students M. Farach, S. Krishnamurthy, F. McFadden, K. W. Ryu, and S. S. Yu for their help. My graduate student R. Kolagotla has kindly supplied almost all of the original figures for this book.

Outside the University of Maryland, there are many people who have reviewed one or more chapters of the book and provided me with their

insightful comments. I would like to thank in particular M. Atallah, R. Giancarlo, G. Landau, V. Pan, J. Reif, and J. Schmidt for their help. I am also indebted to the following reviewers for their many critical comments on an earlier version of the manuscript: S. Baase, L. Heath, U. Manber, B. Moret, S. Seidman, D. Hirschberg, J. Koegel, and J. Zachary.

Finally, I wish to thank my wife S. Laskowski, mostly for her understanding of the time commitment necessary to write such a book, and partly for her editorial comments on several portions of this book.

College Park, Maryland Joseph JáJá

CONTENTS

PREFACE			iii
1	INTRODUCTION		1
	1.1	Parallel Processing	2
	1.2	Background	4
	1.3	Parallel Models	6
	1.4	Performance of Parallel Algorithms	26
	1.5	The Work-Time Presentation Framework of Parallel Algorithms	27
	1.6	The Optimality Notion	32
	1.7	*Communication Complexity	33
	1.8	Summary	35
		Exercises	36
		Bibliographic Notes	40
		References	40
2	BASIC TECHNIQUES		43
	2.1	Balanced Trees	43
	2.2	Pointer Jumping	52
	2.3	Divide and Conquer	56
	2.4	Partitioning	61
	2.5	Pipelining	65
	2.6	Accelerated Cascading	71

	2.7	Symmetry Breaking	75
	2.8	Summary	80
		Exercises	82
		Bibliographic Notes	88
		References	88
3	LISTS AND TREES		91
	3.1	List Ranking	92
	3.2	The Euler-Tour Technique	108
	3.3	Tree Contraction	118
	3.4	Lowest Common Ancestors	128
	3.5	Summary	136
		Exercises	136
		Bibliographic Notes	143
		References	143
4	SEARCHING, MERGING, AND SORTING		145
	4.1	Searching	146
	4.2	Merging	148
	4.3	Sorting	157
	4.4	Sorting Networks	173
	4.5	Selection	181
	4.6	*Lower Bounds for Comparison Problems	184
	4.7	Summary	192
		Exercises	193
		Bibliographic Notes	200
		References	202
5	GRAPHS		203
	5.1	Connected Components	204
	5.2	Minimum Spanning Trees	222
	5.3	Biconnected Components	227
	5.4	Ear Decomposition	240
	5.5	Directed Graphs	246
	5.6	Summary	252
		Exercises	253
		Bibliographic Notes	259
		References	260
6	PLANAR GEOMETRY		263
	6.1	The Convex-Hull Problem Revisited	264
	6.2	Intersections of Convex Sets	272
	6.3	Plane Sweeping	279

	6.4	Visibility Problems	**285**
	6.5	Dominance Counting	**291**
	6.6	Summary	**299**
		Exercises	**300**
		Bibliographic Notes	**307**
		References	**308**
7	STRINGS		**311**
	7.1	Preliminary Facts About Strings	**312**
	7.2	String Matching	**318**
	7.3	Text Analysis	**319**
	7.4	Pattern Analysis	**327**
	7.5	Suffix Trees	**339**
	7.6	Applications of Suffix Trees	**352**
	7.7	Summary	**355**
		Exercises	**357**
		Bibliographic Notes	**363**
		References	**364**
8	ARITHMETIC COMPUTATIONS		**367**
	8.1	Linear Recurrences	**368**
	8.2	Triangular Linear Systems	**375**
	8.3	The Discrete Fourier Transform	**379**
	8.4	Polynomial Multiplication and Convolution	**386**
	8.5	Toeplitz Matrices	**389**
	8.6	Polynomial Division	**394**
	8.7	Polynomial Evaluation and Interpolation	**397**
	8.8	General Dense Matrices	**403**
	8.9	*Dense Structured Matrices	**410**
	8.10	Summary	**419**
		Exercises	**421**
		Bibliographic Notes	**429**
		References	**430**
9	RANDOMIZED ALGORITHMS		**433**
	9.1	Performance Measures of Randomized Parallel Algorithms	**434**
	9.2	The Problem of the Fractional Independent Set	**441**
	9.3	Point Location in Triangulated Planar Subdivisions	**445**
	9.4	Pattern Matching	**450**
	9.5	Verification of Polynomial Identities	**460**
	9.6	Sorting	**464**
	9.7	Maximum Matching	**473**

9.8 Summary **485**
Exercises **485**
Bibliographic Notes **490**
References **491**

10 *LIMITATIONS OF PRAMS **495**
10.1 Simulations Between PRAM Models **496**
10.2 Lower Bounds for the CREW PRAM **502**
10.3 Lower Bounds for the EREW PRAM **508**
10.4 Lower Bounds for the CRCW PRAM **512**
10.5 Introduction to *P*-Completeness **529**
10.6 Summary **553**
Exercises **554**
Bibliographic Notes **558**
References **559**

INDEX **563**

1

Introduction

The purpose of this chapter is to introduce several parallel models and to specify a suitable framework for presenting and analyzing parallel algorithms. A commonly accepted model for designing and analyzing sequential algorithms consists of a central processing unit with a random-access memory attached to it. The typical instruction set for this model includes reading from and writing into the memory, and basic logic and arithmetic operations. The successful model is due to its simplicity and its ability to capture the performance of sequential algorithms on von Neumann-type computers. Unfortunately parallel computation suffers from the lack of such a widely accepted algorithmic model. There is no such model primarily because the performance of parallel algorithms depends on a set of interrelated factors in a complex fashion that is machine dependent. These factors include computational concurrency, processor allocation and scheduling, communication, and synchronization.

In this chapter, we start with a general discussion of parallel processing and related performance measures. We then introduce the models most widely used in algorithm development and analysis. These models are based on directed acyclic graphs, shared memory, and networks. **Directed acyclic graphs** can be used to represent certain parallel computations in a natural

way, and can provide a simple parallel model that does not include any architecture-related features. The **shared-memory model,** where a number of processors communicate through a common global memory, offers an attractive framework for the development of algorithmic techniques for parallel computations. Unlike the two other models, the **network model** captures communication by incorporating the topology of the interconnections into the model itself. We show several parallel algorithms on these models, followed by a brief comparison.

The shared memory model serves as our vehicle for designing and analyzing parallel algorithms in this book and has been a fertile ground for theoretical research into both the power and limitations of parallelism. We shall describe a general framework for presenting and analyzing parallel algorithms in this model.

1.1 Parallel Processing

The main purpose of parallel processing is to perform computations faster than can be done with a single processor by using a number of processors concurrently. The pursuit of this goal has had a tremendous influence on almost all the activities related to computing. The need for **faster solutions** and for **solving larger-size problems** arises in a wide variety of applications. These include fluid dynamics, weather prediction, modeling and simulation of large systems, information processing and extraction, image processing, artificial intelligence, and automated manufacturing.

Three main factors have contributed to the current strong trend in favor of parallel processing. First, the hardware cost has been falling steadily; hence, it is now possible to build systems with many processors at a reasonable cost. Second, the very large scale integration (VLSI) circuit technology has advanced to the point where it is possible to design complex systems requiring millions of transistors on a single chip. Third, the fastest cycle time of a von Neumann-type processor seems to be approaching fundamental physical limitations beyond which no improvement is possible; in addition, as higher performance is squeezed out of a sequential processor, the associated cost increases dramatically. All these factors have pushed researchers into exploring parallelism and its potential use in important applications.

*A **parallel computer** is simply a collection of processors, typically of the same type, interconnected in a certain fashion to allow the coordination of their activities and the exchange of data.* The processors are assumed to be located

within a small distance of one another, and are primarily used to solve a given problem jointly. Contrast such computers with **distributed systems,** where a set of possibly many different types of processors are distributed over a large geographic area, and where the primary goals are to use the available distributed resources, and to collect information and transmit it over a network connecting the various processors.

Parallel computers can be classified according to a variety of architectural features and modes of operations. In particular, these criteria include the type and the number of processors, the interconnections among the processors and the corresponding communication schemes, the overall control and synchronization, and the input/output operations. These considerations are outside the scope of this book.

Our main goal is to present algorithms that are suitable for implementation on parallel computers. We emphasize techniques, paradigms, and methods, rather than detailed algorithms for specific applications. An immediate question comes to mind: How should an algorithm be evaluated for its suitability for parallel processing? As in the case of sequential algorithms, there are several important criteria, such as time performance, space utilization, and programmability. The situation for parallel algorithms is more complicated due to the presence of additional parameters, such as the number of processors, the capacities of the local memories, the communication scheme, and the synchronization protocols. To get started, we introduce two general measures commonly used for evaluating the performance of a parallel algorithm.



Let P be a given computational problem and let n be its input size. Denote the sequential complexity of P by $T^*(n)$. That is, there is a sequential algorithm that solves P within this time bound, and, in addition, we can prove that no sequential algorithm can solve P faster. Let A be a parallel algorithm that solves P in time $T_p(n)$ on a parallel computer with p processors. Then, the **speedup** achieved by A is defined to be

$$S_p(n) = \frac{T^*(n)}{T_p(n)}.$$

Clearly, $S_p(n)$ measures the speedup factor obtained by algorithm A when p processors are available. Ideally, since $S_p(n) \leq p$, we would like to design algorithms that achieve $S_p(n) \approx p$. In reality, there are several factors that introduce inefficiencies. These include insufficient concurrency in the computation, delays introduced by communication, and overhead incurred in synchronizing the activities of various processors and in controlling the system.

Note that $T_1(n)$, the running time of the parallel algorithm A when the number p of processors is equal to 1, is not necessarily the same as $T^*(n)$; hence, the speedup is measured relative to the *best possible sequential algorithm.* It is common practice to replace $T^*(n)$ by the time bound of the best *known* sequential algorithm whenever the complexity of the problem is not known.

Another performance measure of the parallel algorithm A is **efficiency,** defined by

$$E_p(n) = \frac{T_1(n)}{pT_p(n)}.$$

This measure provides an indication of the effective utilization of the p processors relative to the given algorithm. A value of $E_p(n)$ approximately equal to 1, for some p, indicates that algorithm A runs approximately p times faster using p processors than it does with one processor. It follows that each of the processors is doing "useful work" during each time step relative to the total amount of work required by algorithm A.

There exists a limiting bound on the running time, denoted by $T_\infty(n)$, beyond which the algorithm cannot run any faster, no matter what the number of processors. Hence, $T_p(n) \geq T_\infty(n)$, for any value of p, and thus the efficiency $E_p(n)$ satisfies $E_p(n) \leq T_1(n)/pT_\infty(n)$. Therefore, the efficiency of an algorithm degrades quickly as p grows beyond $T_1(n)/T_\infty(n)$.

Our main goal in this book is to develop parallel algorithms that can provably achieve the best possible speedup. Therefore, our model of parallel computation must allow the mathematical derivation of an estimate on the running time $T_p(n)$ and the establishment of lower bounds on the best possible speedup for a given problem. Before introducing several candidate models, we outline the background knowledge that readers should have.

1.2 Background

Readers should have an understanding of elementary data structures and basic techniques for designing and analyzing sequential algorithms. Such material is usually covered at the undergraduate level in computer science and computer engineering curricula. Our terminology and notation are standard; they are described in several of the references given at the end of this chapter.

Algorithms are expressed in a high-level language in common use. Each algorithm begins with a description of its input and its output, followed by a statement (which consists of a sequence of one or more statements). We next give a list of the statements most frequently used in our algorithms. We shall augment this list later with constructs needed for expressing parallelism.

1. *Assignment statement:*

   variable: = expression

   The expression on the right is evaluated and assigned to the variable on the left.

2. *Begin/end statement:*

   begin
   statement
   statement
   ⋮
   statement
   end

   This block defines a sequence of statements that must be executed in the order in which they appear.

3. *Conditional statement*:

   if (condition) **then** statement [**else** statement]

   The condition is evaluated, and the statement following **then** is executed if the value of the condition is **true.** The **else** part is optional; it is executed if the condition is **false.** In the case of nested conditional statements, we use braces to indicate the **if** statement associated with each **else** statement.

4. *Loops*: We use one of the following two formats:

   for variable = initial value **to** final value **do** statement
   while (condition) **do** statement

   The interpretation of the **for** loop is as follows. If the initial value is less than or equal to the final value, the statement following **do** is executed, and the value of the variable is incremented by one. Otherwise, the execution of the loop terminates. The same process is repeated with the new value of the variable, until that value exceeds the final value, in which case the execution of the loop terminates.

   The **while** loop is similar, except that the condition is tested before each execution of the statement. If the condition is **true,** the statement is executed; otherwise, the execution of the loop terminates.

5. *Exit statement*:

   exit

   This statement causes the execution of the whole algorithm to terminate.

The bounds on the resources (for example, time and space) required by a sequential algorithm are measured as a function of the **input size,** which reflects the amount of data to be processed. We are primarily interested in the **worst-case** analysis of algorithms; hence, given an input size n, each resource bound represents the maximum amount of that resource required by any instance of size n. These bounds are expressed **asymptotically** using the following standard notation:

- $T(n) = O(f(n))$ if there exist positive constants c and n_0 such that $T(n) \leq cf(n)$, for all $n \geq n_0$.

- $T(n) = \Omega(f(n))$ if there exist positive constants c and n_0 such that $T(n) \geq cf(n)$, for all $n \geq n_0$.
- $T(n) = \Theta(f(n))$ if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$.

The **running time** of a sequential algorithm is estimated by the number of *basic operations* required by the algorithm as a function of the input size. This definition naturally leads to the questions of what constitutes a basic operation, and whether the cost of an operation should be a function of the word size of the data involved. These issues depend on the specific problem at hand and the model of computation used. Briefly, we charge a unit of time to the operations of reading from and writing into the memory, and to basic arithmetic and logic operations (such as adding, subtracting, comparing, or multiplying two numbers, and computing the bitwise logic OR or AND of two words). The cost of an operation does not depend on the word size; hence, we are using what is called the **uniform cost criterion.** A formal computational model suitable for our purposes is the **Random Access Machine (RAM),** which assumes the presence of a central processing unit with a random-access memory attached to it, and some way to handle the input and the output operations. A knowledge of this model beyond our informal description is not necessary for understanding the material covered in this book. For more details concerning the analysis of algorithms, refer to the bibliographic notes at the end of this chapter.

Finally, all *logarithms* used in this book are to the base 2 unless otherwise stated. A logarithm used in an asymptotic expression will always have a minimum value of 1.

1.3 Parallel Models

The RAM model has been used successfully to predict the performance of sequential algorithms. Modeling parallel computation is considerably more challenging given the new dimension introduced by the presence of many interconnected processors. We should state at the outset that we are primarily interested in *algorithmic models that can be used as general frameworks for describing and analyzing parallel algorithms*. Ideally, we would like our model to satisfy the following (conflicting) requirements:

- **Simplicity:** The model should be simple enough to allow us to describe parallel algorithms easily, and to analyze mathematically important performance measures such as speed, communication, and memory utilization. In addition, the model should not be tied to any particular class of architectures, and hence should be as hardware-independent as possible.

- **Implementability:** The parallel algorithms developed for the model should be easily implementable on parallel computers. In addition, the analysis performed should capture in a significant way the actual performance of these algorithms on parallel computers.

Thus far, no single algorithmic parallel model has proved to be acceptable to most researchers in parallel processing. The literature contains an abundant number of parallel algorithms for specific architectures and specific parallel machines; such reports describe a great number of case studies, but offer few unifying techniques and methods.

In Sections 1.3.1 through 1.3.3, we introduce three parallel models and briefly discuss their relative merits. Other parallel models, such as **parallel comparison trees, sorting networks,** and **Boolean circuits,** will be introduced later in the book. We also state our choice of the parallel model used in this book and provide justification for this choice.

1.3.1 DIRECTED ACYCLIC GRAPHS

Many computations can be represented by **directed acyclic graphs (dags)** in a natural way. Each input is represented by a node that has no incoming arcs. Each operation is represented by a node that has incoming arcs from the nodes representing the operands. The indegree of each internal node is at most two. A node whose outdegree is equal to zero represents an output. We assume the unit cost criterion, where each node represents an operation that takes one unit of time.

A directed acyclic graph with n input nodes represents a computation that has no branching instructions and that has an input of size n. Therefore, an algorithm is represented by a family of dags $\{G_n\}$, where G_n corresponds to the algorithm with input size n.

This model is particularly suitable for analyzing numerical computations, since branching instructions are typically used to execute a sequence of operations a certain number of times, dependent on the input size n. In this case, we can unroll a branching instruction by duplicating the sequence of operations to be repeated the appropriate number of times.

A dag specifies the operations performed by the algorithm, and implies **precedence constraints** on the order in which these operations must be performed. It is completely architecture-independent.

EXAMPLE 1.1:

Consider the problem of computing the sum S of the $n = 2^k$ elements of an array A. Two possible algorithms are represented by their dags in Fig. 1.1 for $n = 8$. The algorithm in Fig. 1.1(a) computes the partial sums consecutively, starting with $A(1) + A(2)$, followed by $(A(1) + A(2)) + A(3)$, and so on. The

algorithm in Fig. 1.1(b) proceeds in a complete binary tree fashion that begins by computing the sums $A(1) + A(2), A(3) + A(4), \ldots, A(n-1) + A(n)$ at the lowest level, and repeats the process at the next level with $\frac{n}{2}$ elements, and so on until the sum is computed at the root. □

The dag model can be used to analyze the performance of a parallel algorithm under the assumption that *any processor can access the data computed by any other processor, without incurring additional cost.* We can specify a particular implementation of the algorithm by **scheduling** each node for execution on a particular processor. More precisely, given p processors, we have to associate with each internal node i of the dag a pair (j_i, t_i), where $j_i \leq p$ is the index of a processor and t_i is a time unit (processor P_{j_i} executes the operation specified by node i at time t_i), such that the following two conditions hold:

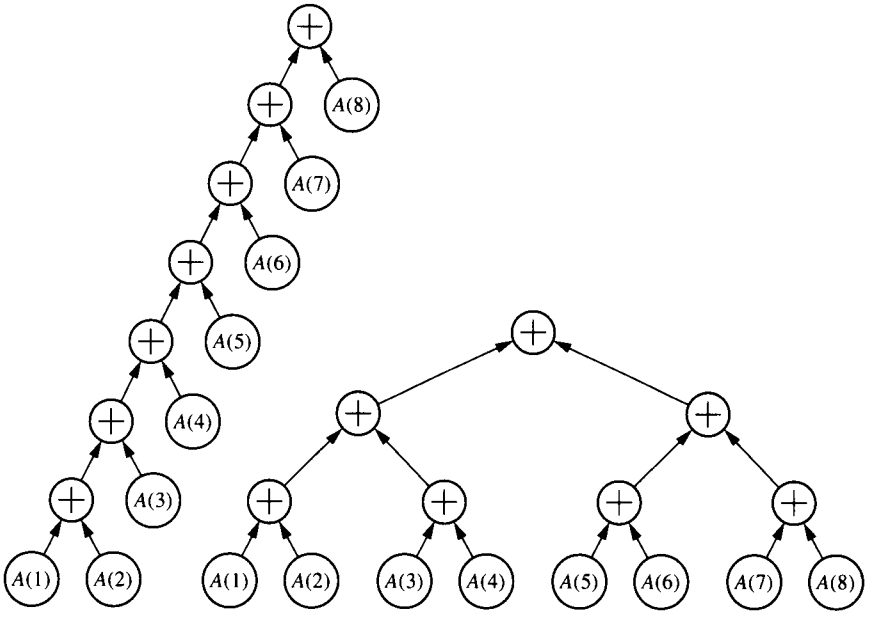


FIGURE 1.1
The dags of two possible algorithms for Example 1.1. (a) A dag for computing the sum of eight elements. (b) An alternate dag for computing the sum of eight elements based on a balanced binary tree scheme.

1. If $t_i = t_k$ for some $i \neq k$, then $j_i \neq j_k$. That is, each processor can perform a single operation during each unit of time.
2. If (i, k) is an arc in the graph, then $t_k \geq t_i + 1$. That is, the operation represented by node k should be scheduled after the operation represented by node i has been completed.

The time t_i of an input node i is assumed to be 0, and no processor is allocated to the node i. We call the sequence $\{(j_i, t_i) \mid i \in N\}$ a **schedule** for the parallel execution of the dag by p processors, where N is the set of nodes in the dag.

For any given schedule, the corresponding time for executing the algorithm is given by $\max_{i \in N} t_i$. The **parallel complexity** of the dag is defined by $T_p(n) = \min\{\max_{i \in N} t_i\}$, where the minimum is taken over all schedules that use p processors. Clearly, the depth of the dag, which is the length of the longest path between an input and an output mode, is a lower bound on $T_p(n)$, for any number p of processors.

EXAMPLE 1.2:

Consider the two sum algorithms presented in Example 1.1 for an arbitrary number n of elements. It is clear that the best schedule of the algorithm represented in Fig. 1.1(a) takes $O(n)$ time, regardless of the number of processors available, whereas the best schedule of the dag of Fig. 1.1(b) takes $O(\log n)$ time with $\frac{n}{2}$ processors. In either case, the scheduling algorithm is straightforward and proceeds bottom up, level by level, where all the nodes at the same level have the same execution time. □

EXAMPLE 1.3: (Matrix Multiplication)

Let A and B be two $n \times n$ matrices. Consider the standard algorithm to compute the product $C = AB$. Each $C(i, j)$ is computed using the expression $C(i,j) = \sum_{l=1}^{n} A(i, l)B(l, j)$. A dag to compute $C(i, j)$ for $n = 4$ is shown in Fig. 1.2. Given n^3 processors, the operations can be scheduled level by level, using n processors to compute each entry of C; hence, the dag can be scheduled to compute C in $O(\log n)$ time. □

1.3.2 THE SHARED-MEMORY MODEL

The next model is a natural extension of our basic sequential model; in the new model, many processors have access to a single shared memory unit. More precisely, the shared-memory model consists of a number of processors, each of which has its own local memory and can execute its own local program, and all of which communicate by exchanging data through a shared

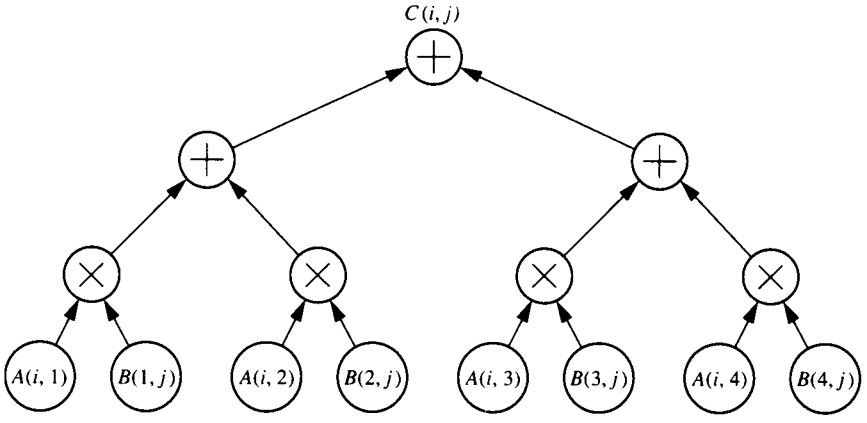


FIGURE 1.2
A dag for computing an entry $C(i, j)$ of the matrix product $C = AB$ for the case of 4×4 matrices.

memory unit. Each processor is uniquely identified by an index, called a **processor number** or **processor id,** which is available locally (and hence can be referred to in the processor's program). Figure 1.3 shows a general view of a shared-memory model with p processors. These processors are indexed 1, 2, ... , p. Shared memory is also referred to as **global memory.**

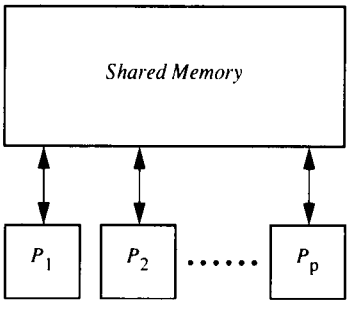


FIGURE 1.3
The shared-memory model.

There are two basic modes of operation of a shared-memory model. In the first mode, called **synchronous,** *all the processors operate synchronously under the control of a common clock.* A standard name for the synchronous shared-memory model is the **parallel random-access machine (PRAM) model.** In the second mode, called **asynchronous,** *each processor operates under a separate clock.* In the asynchronous mode of operation, it is the programmer's responsibility to set appropriate synchronization points whenever necessary. More precisely, if data need to be accessed by a processor, it is the programmer's responsibility to ensure that the correct values are obtained, since the value of a shared variable is determined dynamically during the execution of the programs of the different processors.

Since each processor can execute its own local program, our shared-memory model is a **multiple instruction multiple data (MIMD) type.** That is, each processor may execute an instruction or operate on data different from those executed or operated on by any other processor during any given time unit. For a given algorithm, the size of data transferred between the shared memory and the local memories of the different processors represents the amount of **communication** required by the algorithm.

Before introducing the next example, we augment our algorithmic language with the following two constructs:

global read(X, Y)
global write(U, V)

The effect of the **global read** instruction is to move the block of data X stored in the shared memory into the local variable Y. Similarly, the effect of the **global write** is to write the local data U into the shared variable V.

EXAMPLE 1.4: (Matrix Vector Multiplication on the Shared-Memory Model)

Let A be an $n \times n$ matrix, and let x be a vector of order n, both stored in the shared memory. Assume that we have $p \leq n$ processors such that $r = n/p$ is an integer and that the mode of operation is *asynchronous*. Let A be partitioned as follows:

$$A = \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_p \end{bmatrix},$$

where each block A_i is of size $r \times n$. The problem of computing the product $y = Ax$ can be solved as follows. Each processor P_i reads A_i and x from the shared memory, then performs the computation $z = A_i x$, and finally stores the r components of z in the appropriate components of the shared variable y.

In the algorithm that follows, we use $A(l:u, s:t)$ to denote the submatrix of A consisting of rows $l, l+1, \ldots, u$ and columns $s, s+1, \ldots, t$. The same notation can be used to indicate a subvector of a given vector. Each processor executes the same algorithm.

ALGORITHM 1.1

(Matrix Vector Multiplication on the Shared-Memory Model)

Input: *An $n \times n$ matrix A and a vector x of order n residing in the shared memory. The initialized local variables are (1) the order n, (2) the processor number i, and (3) the number $p \leq n$ of processors such that $r = n/p$ is an integer.*

Output: *The components $(i-1)r+1, \ldots, ir$ of the vector $y = Ax$ stored in the shared variable y.*

begin

1. **global read**(x, z)
2. **global read**$(A((i-1)r+1{:}ir, 1:n), B)$
3. *Compute* $w = Bz$.
4. **global write**$(w, y((i-1)r+1:ir))$

end

Notice that a *concurrent read* of the same shared variable x is required by all the processors at step 1. However, no two processors attempt to write into the same location of the shared memory.

We can estimate the amount of computation and communication used by the algorithm as follows. Step 3 is the only computation step that requires $O(n^2/p)$ arithmetic operations. Steps 1 and 2 transfer $O(n^2/p)$ numbers from the shared memory into each processor, and step 4 stores n/p numbers from each local memory in the shared memory.

An important feature of Algorithm 1.1 is that the processors do not need to synchronize their activities, given the way the matrix vector product was partitioned. On the other hand, we can design a parallel algorithm based on partitioning A and x into p blocks such that $A = (A_1, A_2, \ldots, A_p)$ and $x = (x_1, x_2, \ldots, x_p)$, where each A_i is of size $n \times r$ and each x_i is of size r. The product $y = Ax$ is now given by $y = A_1x_1 + A_2x_2 + \cdots + A_rx_r$. Hence, processor P_i can compute $z_i = A_ix_i$ after reading A_i and x_i from the shared memory, for $1 \leq i \leq p$. At this point, however, no processor should begin the computation of the sum $z_1 + z_2 + \cdots + z_r$ before ensuring that all the processors have completed their matrix vector products. Therefore, an explicit synchronization primitive must be placed in each processor's program after the computation of $z_i = A_ix_i$ to force all the processors to synchronize before continuing the execution of their programs. □

In the remainder of this section, we concentrate on the PRAM model, which is the synchronous shared-memory model.

Algorithms developed for the PRAM model have been of type **single instruction multiple data (SIMD).** That is, all processors execute the same program such that, during each time unit, all the active processors are executing the same instruction, but with different data in general. However, as the model stands, we can load different programs into the local memories of the processors, as long as the processors can operate synchronously; hence, different types of instructions can be executed within the unit time allocated for a step.

EXAMPLE 1.5: (Sum on the PRAM)

Given an array A of $n = 2^k$ numbers, and a PRAM with n processors $\{P_1, P_2, \ldots, P_n\}$, we wish to compute the sum $S = A(1) + A(2) + \ldots + A(n)$. Each processor executes the same algorithm, given here for processor P_i.

ALGORITHM 1.2
(Sum on the PRAM Model)
Input: *An array A of order $n = 2^k$ stored in the shared memory of a PRAM with n processors. The initialized local variables are n and the processor number i.*
Output: *The sum of the entries of A stored in the shared location S. The array A holds its initial value.*

```
begin
  1. global read(A(i), a)
  2. global write(a, B(i))
  3. for h = 1 to log n do
        if (i ≤ n/2^h) then
          begin
                global read(B(2i − 1), x)
                global read(B(2i), y)
                Set z: = x + y
                global write(z, B(i))
          end
  4. if i = 1 then global write(z, S)
end
```

Figure 1.4 illustrates the algorithm for the case when $n = 8$. During steps 1 and 2, a copy B of A is created and is stored in the shared memory. The computation scheme (step 3) is based on a balanced binary tree whose leaves correspond to the elements of A. The processor responsible for performing

an operation is indicated below the node representing the operation. Note that P_1, which is responsible for updating the value of $B(1)$ and for writing the sum into S, is always active during the execution of the algorithm, whereas P_5, P_6, P_7, and P_8 are active only during steps 1 and 2.

Using this example, we emphasize the following key assumptions about the PRAM model.

- **Shared-memory:** The arrays A and B are stored in the global memory and can be accessed by any processor.
- **Synchronous mode of operation:** In each unit of time, each processor is allowed to execute an instruction or to stay idle. Note that the condition of the **if** statement in the loop defined in step 3 is satisfied by only some processors. Each of the remaining processors stays idle during that time. □

There are several variations of the PRAM model based on the assumptions regarding the handling of the simultaneous access of several processors to the same location of the global memory. The **exclusive read exclusive write**

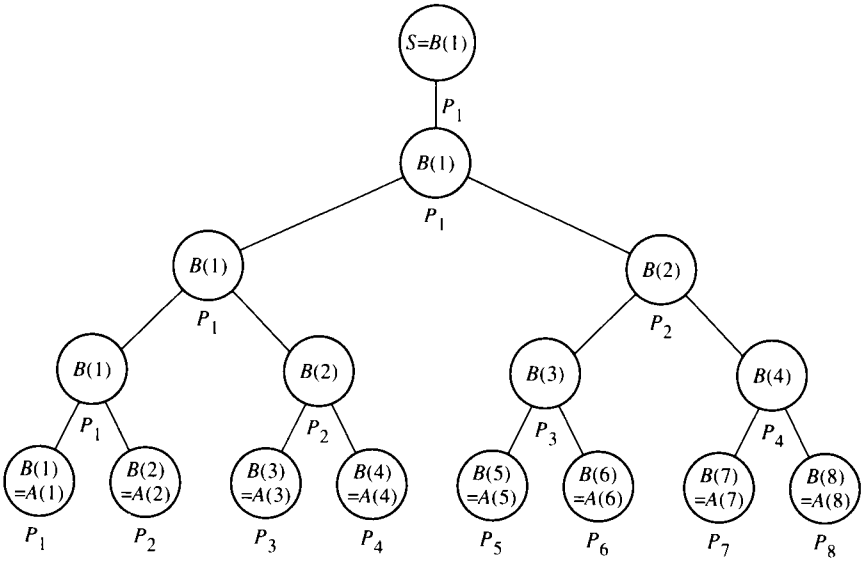


FIGURE 1.4
Computation of the sum of eight elements on a PRAM with eight processors. Each internal node represents a sum operation. The specific processor executing the operation is indicated below each node.

(**EREW**) PRAM does not allow any simultaneous access to a single memory location. The **concurrent read exclusive write** (**CREW**) PRAM allows simultaneous access for a read instruction only. Access to a location for a read or a write instruction is allowed in the **concurrent read concurrent write** (**CRCW**) PRAM. The three principal varieties of CRCW PRAMs are differentiated by how concurrent writes are handled. The **common** CRCW PRAM allows concurrent writes only when all processors are attempting to write the same value. The **arbitrary** CRCW PRAM allows an arbitrary processor to succeed. The **priority** CRCW PRAM assumes that the indices of the processors are linearly ordered, and allows the one with the minimum index to succeed. Other variations of the CRCW PRAM model exist. It turns out that these three models (EREW, CREW, CRCW) do not differ substantially in their computational powers, although the CREW is more powerful than the EREW, and the CRCW is most powerful. We discuss their relative powers in Chapter 10.

Remark 1.1: To simplify the presentation of PRAM algorithms, *we omit the details concerning the memory-access operations*. An instruction of the form *Set A:* $= B + C$, where A, B, and C are shared variables, should be interpreted as the following sequence of instructions.

global read(B, x)
global read(C, y)
Set $z := x + y$
global write(z, A)

In the remainder of this book, no PRAM algorithm will contain explicit memory-access instructions. □

EXAMPLE 1.6: (Matrix Multiplication on the PRAM)

Consider the problem of computing the product C of the two $n \times n$ matrices A and B, where $n = 2^k$, for some integer k. Suppose that we have n^3 processors available on our PRAM, denoted by $P_{i,j,l}$, where $1 \leq i, j, l \leq n$. Processor $P_{i,j,l}$ computes the product $A(i,l)B(l,j)$ in a single step. Then, for each pair (i,j), the n processors $P_{i,j,l}$, where $1 \leq l \leq n$, compute the sum $\sum_{l=1}^{n} A(i,l)B(l,j)$ as described in Example 1.5.

The algorithm for processor $P_{i,j,l}$ is stated next (recall Remark 1.1).

ALGORITHM 1.3

(Matrix Multiplication on the PRAM)

Input: *Two $n \times n$ matrices A and B stored in the shared memory, where $n = 2^k$. The initialized local variables are n, and the triple of indices (i, j, l) identifying the processor.*

Output: *The product* $C = AB$ *stored in the shared memory.*
begin
1. *Compute* $C'(i,j,l) = A(i,l)B(l,j)$
2. **for** $h = 1$ **to** $\log n$ **do**
   if $(l \leq n/2^h)$ **then** *set* $C'(i,j,l) := C'(i,j,2l-1) + C'(i,j,2l)$
3. **if** $(l = 1)$ **then** *set* $C(i,j) := C'(i,j,1)$

end

Notice that the previous algorithm requires concurrent read capability, since different processors may have to access the same data while executing step 1. For example, processors $P_{i,1,l}, P_{i,2,l}, \ldots, P_{i,n,l}$ all require $A(i,l)$ from the shared memory during the execution of step 1. Hence, this algorithm runs on the CREW PRAM model. As for the running time, the algorithm takes $O(\log n)$ parallel steps. □

Remark 1.2: If we modify step 3 of Algorithm 1.3 by removing the **if** condition (that is, replacing the whole statement by *Set* $C(i, j) := C'(i, j, 1)$), then the corresponding algorithm requires a concurrent write of the same value capability. In fact, processors $P_{i,j,1}, P_{i,j,2}, \ldots, P_{i,j,n}$ all attempt to write the value $C'(i,j,1)$ into location $C(i,j)$. □

1.3.3 THE NETWORK MODEL

A **network** can be viewed as a graph $G = (N, E)$, where each node $i \in N$ represents a processor, and each edge $(i,j) \in E$ represents a two-way communication link between processors i and j. Each processor is assumed to have its own local memory, and no shared memory is available. As in the case of the shared-memory model, the operation of a network may be either **synchronous** or **asynchronous.**

In describing algorithms for the network model, we need additional constructs for describing communication between the processors. We use the following two constructs:

send(X, i)
receive(Y, j)

A processor P executing the **send** instruction sends a copy of X to processor P_i, then resumes the execution of the next instruction immediately. A processor P executing the **receive** instruction suspends the execution of its program until the data from processor P_j are received. It then stores the data in Y and resumes the execution of its program.

The processors of an asynchronous network coordinate their activities by exchanging messages, a scheme referred to as the **message-passing model.** Note that, in this case, a pair of communicating processors do not necessarily have to be adjacent; the process of delivering each message from its source to its destination is called **routing.** The study of routing algorithms is outside the scope of this book; see the bibliographic notes at the end of this chapter for references.

The network model incorporates the topology of the interconnection between the processors into the model itself. There are several parameters used to evaluate the topology of a network G. Briefly, these include the **diameter,** which is the maximum distance between any pair of nodes, the **maximum degree** of any node in G, and **the node** or **edge connectivity** of G.

We shall introduce the following representative topologies: the linear array, the two-dimensional mesh, and the hypercube. Many other networks have been studied extensively for their suitability for parallel processing; they are not mentioned here. Several books describing them are given in the list of references at the end of this chapter.

The Linear Processor Array and the Ring. The **linear processor array** consists of p processors $P_1, P_2, \ldots, P_p$ connected in a linear array; that is, processor P_i is connected to P_{i-1} and to P_{i+1}, whenever they exist. Figure 1.5 shows a linear array with eight processors. The diameter of the linear array is $p - 1$; its maximum degree is 2.

A **ring** is a linear array of processors with an end-around connection; that is, processors P_1 and P_p are directly connected.

EXAMPLE 1.7: (Matrix Vector Product on a Ring)

Given an $n \times n$ matrix A and a vector x of order n, consider the problem of computing the matrix vector product $y = Ax$ on a ring of p processors, where $p \leq n$. Assume that p divides n evenly, and let $r = n/p$. Let A and x be partitioned into p blocks as follows: $A = (A_1, A_2, \cdots, A_p)$ and $x = (x_1, x_2, \ldots, x_p)$, where each A_i is of size $n \times r$ and each x_i is of size r. We can determine the product $y = Ax$ by computing $z_i = A_i x_i$, for $1 \leq i \leq p$, and then accumulating the sum $z_1 + z_2 + \cdots + z_p$.



FIGURE 1.5
A linear array of eight processors.

Let processor P_i of our network hold initially $B = A_i$ and $w = x_i$ in its local memory, for all $1 \leq i \leq p$. Then each processor can compute locally the product Bw, and we can accumulate the sum of these vectors by circulating the partial sums through the ring clockwise. The output vector will be stored in P_1. The algorithm to be executed by each processor is given next.

ALGORITHM 1.4

(Asynchronous Matrix Vector Product on a Ring)

Input: *(1) The processor number i; (2) the number p of processors; (3) the ith submatrix* $B = A(1 : n, (i - 1)r + 1 : ir)$ *of size* $n \times r$, *where* $r = n/p$; *(4) the ith subvector* $w = x((i - 1)r + 1 : ir)$ *of size r.*

Output: *Processor* P_i *computes the vector* $y = A_1x_1 + \cdots + A_ix_i$ *and passes the result to the right. When the algorithm terminates,* P_1 *will hold the product Ax.*

begin

1. *Compute the matrix vector product* $z = Bw$.
2. **if** $i = 1$ **then** *set* $y := 0$
   else receive(y, *left*)
3. *Set* $y := y + z$
4. **send**(y, *right*)
5. **if** $i = 1$ **then receive**(y, *left*)

end

Each processor P_i begins by computing A_ix_i; it stores the resulting vector in the local variable z. At step 2, processor P_1 initializes the vector y to 0, whereas each of the other processors suspends the execution of its program waiting to receive data from its left neighbor. Processor P_1 sets $y = A_1x_1$ and sends the result to the right neighbor in steps 3 and 4, respectively. At this time, P_2 receives A_1x_1 and resumes the execution of its program by computing $A_1x_1 + A_2x_2$ at step 3; it then sends the new vector to the right at step 4. When the executions of all the programs terminate, P_1 holds the product $y = Ax$.

The computation performed by each processor consists of the two operations in steps 1 and 3; hence, the algorithm's computation time is $O(n^2/p)$—say $T_{comp} = \alpha(n^2/p)$, where α is some constant. On the other hand, processor P_1 has to wait until the partial sum $A_1x_1 + \cdots + A_px_p$ has been accumulated before it can execute step 5. Therefore, the total communication time T_{comm} is proportional to the product $p \cdot comm(n)$, where $comm(n)$ is the time it takes to transmit n numbers between adjacent processors. This value can be approximated by $comm(n) = \sigma + n\tau$, where σ is the startup time for transmission, and τ is the rate at which the message can be transferred.

The total execution time is given by $T = T_{comp} + T_{comm} = \alpha(n^2/p) + p(\sigma + n\tau)$. Clearly, a tradeoff exists between the computation time and the

communication time. In particular, the sum is minimized when $\alpha(n^2/p) = p(\sigma + n\tau)$, and hence $p = n\sqrt{\alpha/(\sigma + n\tau)}$. □

The Mesh. The **two-dimensional mesh** is a two-dimensional version of the linear array. It consists of $p = m^2$ processors arranged into an $m \times m$ grid such that processor $P_{i,j}$ is connected to processors $P_{i\pm1,j}$ and $P_{i,j\pm1}$, whenever they exist. Figure 1.6 shows a 4×4 mesh.

The diameter of a $(p = m^2)$-processor mesh is $\sqrt{p}$, and the maximum degree of any node is 4. This topology has several attractive features, such as simplicity, regularity, and extensibility. In addition, it matches the computation structures arising in many applications. However, since the mesh has diameter $2\sqrt{p} - 2$, almost any nontrivial computation requires $\Omega(\sqrt{p})$ parallel steps.

EXAMPLE 1.8: (Systolic Matrix Multiplication on the Mesh)

The problem of computing the product $C = AB$ on an $n \times n$ mesh, where A, B, and C are $n \times n$ matrices, can be solved in $O(n)$ time. Figure 1.7 shows one possible scheme, following the **systolic paradigm,** where the rows of A are fed *synchronously* into the left side of the mesh in a skewed fashion, and the columns of B are fed *synchronously* into the top boundary of the mesh in a skewed fashion. When $P_{i,j}$ receives the two inputs $A(i, l)$ and $B(l, j)$, it performs the operation $C(i,j) := C(i,j) + A(i,l)B(l,j)$; it then sends $A(i,l)$ to its right neighbor and $B(l,j)$ to the neighbor just below it.

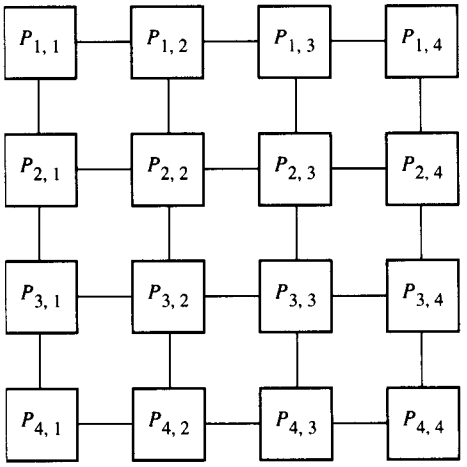


FIGURE 1.6
A 4×4 mesh.

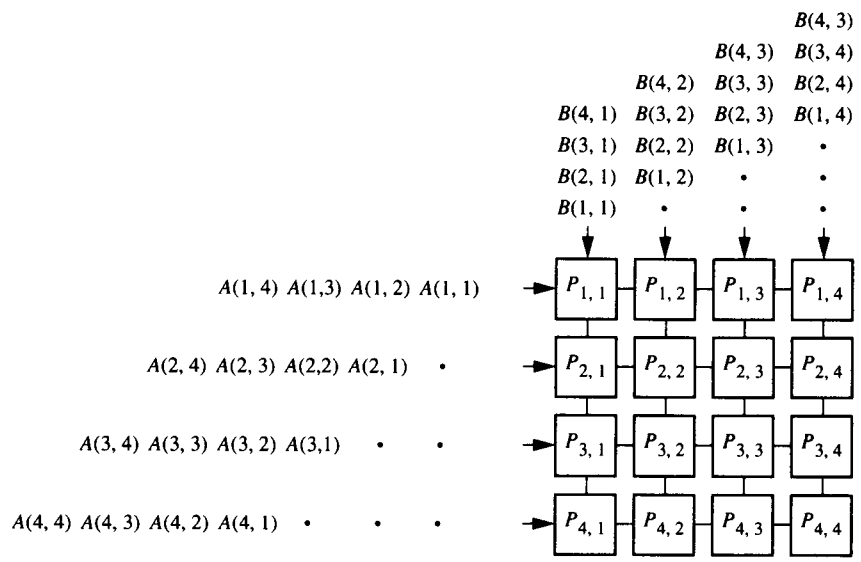


FIGURE 1.7
Matrix multiplication on the mesh using a systolic algorithm. Rows of A move synchronously into the left side, while columns of B move synchronously at the same rate into the top side. When $A(i, l)$ and $B(l, j)$ are available at processor $P_{i,j}$, the operation $C(i, j) = C(i, j) + A(i, l)B(l, j)$ takes place, $A(i, l)$ is sent to $P_{i,j+1}$ (if it exists), and $B(l, j)$ is sent to $P_{i+1,j}$ (if it exists).

After $O(n)$ steps, each processor $P_{i,j}$ will have the correct value of $C(i,j)$. Hence, the algorithm achieves an optimal speedup for n^2 processors relative to the standard matrix-multiplication algorithm, which requires $O(n^3)$ operations.

Systolic algorithms operate in a fully synchronous fashion, where, at each time unit, a processor receives data from some neighbors, then performs some local computation, and finally sends data to some of its neighbors. □

You should not conclude from Example 1.8 that mesh algorithms are typically synchronous. Many of the algorithms developed for the mesh have been asynchronous.

The Hypercube. A **hypercube** consists of $p = 2^d$ processors interconnected into a d-dimensional Boolean cube that can be defined as follows. Let the binary representation of i be $i_{d-1}i_{d-2} \cdots i_0$, where $0 \leq i \leq p - 1$. Then processor P_i is connected to processors $P_{i^{(j)}}$, where $i^{(j)} = i_{d-1} \cdots \bar{i}_j \cdots i_0$,

and $\bar{i}_j = 1 - i_j$, for $0 \leq j \leq d - 1$. In other words, two processors are connected if and only if their indices differ in only one bit position. Notice that our processors are indexed from 0 to $p - 1$.

The hypercube has a recursive structure. We can extend a d-dimensional cube to a $(d + 1)$-dimensional cube by connecting corresponding processors of two d-dimensional cubes. One cube has the most significant address bit equal to 0; the other cube has the most significant address bit equal to 1. Figure 1.8 shows a four-dimensional hypercube.

The diameter of a d-dimensional hypercube is $d = \log p$, since the distance between any two processors P_i and P_j is equal to the number of bit positions in which i and j differ; hence, it is less than or equal to d, and the distance between say P_0 and P_{2^d-1} is d. Each node is of degree $d = \log p$.

The hypercube is popular because of its regularity, its small diameter, its many interesting graph-theoretic properties, and its ability to handle many computations quickly and simply.

We next develop *synchronous hypercube algorithms* for several simple problems, including matrix multiplication.

EXAMPLE 1.9: (Sum on the Hypercube)

Each entry $A(i)$ of an array A of size n is stored initially in the local memory of processor P_i of an $(n = 2^d)$-processor synchronous hypercube. The goal is

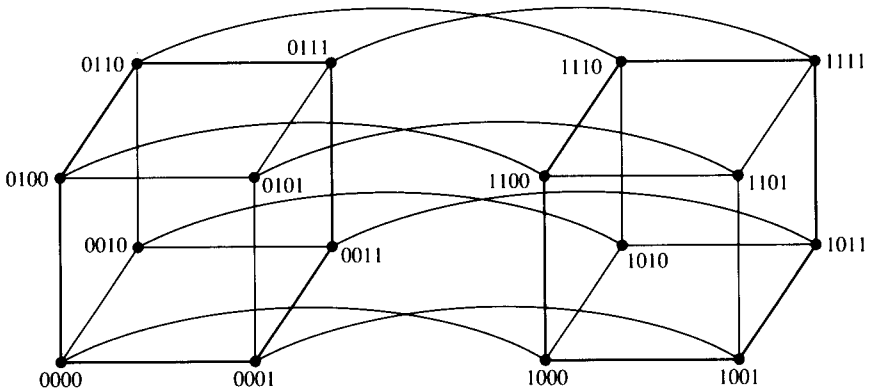


FIGURE 1.8
A four-dimensional hypercube, where the index of each processor is given in binary. Two processors are connected if and only if their indices differ in exactly one bit position.

to compute the sum $S = \sum_{i=0}^{n-1} A(i)$, and to store it in processor P_0. Notice that the indices of the array elements begin with 0.

The algorithm to compute S is straightforward. It consists of d iterations. The first iteration computes sums of pairs of elements between processors whose indices differ in the most significant bit position. These sums are stored in the $(d-1)$-dimensional subcube whose most significant address bit is equal to 0. The remaining iterations continue in a similar fashion.

In the algorithm that follows, the hypercube operates synchronously, and $i^{(l)}$ denotes the index i whose l bit has been complemented. The instruction $A(i) := A(i) + A(i^{(l)})$ involves two substeps. In the first substep, P_i copies $A(i^{(l)})$ from processor $P_{i^{(l)}}$ along the link connecting $P_{i^{(l)}}$ and P_i; in the second substep, P_i performs the addition $A(i) + A(i^{(l)})$, storing the result in $A(i)$.

ALGORITHM 1.5
(Sum on the Hypercube)

Input: *An array A of $n = 2^d$ elements such that $A(i)$ is stored in the local memory of processor P_i of an n-processor synchronous hypercube, where $0 \le i \le n-1$.*
Output: *The sum $S = \sum_{i=0}^{n-1} A(i)$ stored in P_0.*
Algorithm for Processor P_i

```
begin
  for l = d − 1 to 0 do
      if (0 ≤ i ≤ 2^l − 1) then
        Set A(i): = A(i) + A(i^(l))
end
```

Consider, for example, the case when $n = 8$. Then, during the first iteration of the **for** loop, the sums $A(0) = A(0) + A(4)$, $A(1) = A(1) + A(5)$, $A(2) = A(2) + A(6)$, and $A(3) = A(3) + A(7)$ are computed and stored in the processors P_0, P_1, P_2, and P_3, respectively. At the completion of the second iteration, we obtain $A(0) = (A(0) + A(4)) + (A(2) + A(6))$ and $A(1) = (A(1) + A(5)) + (A(3) + A(7))$. The third iteration clearly sets $A(0)$ to the sum S. Algorithm 1.5 terminates after $d = \log n$ parallel steps. Compare this algorithm with the PRAM algorithm (Algorithm 1.2) that computes the sum in $O(\log n)$ steps as well. □

EXAMPLE 1.10: (Broadcasting from One Processor on the Hypercube)

Consider the problem of broadcasting an item X held in the register $D(0)$ of P_0 to all the processors P_i of a p-processor hypercube, where $p = 2^d$.

A simple strategy can be used to solve this problem. We proceed from the lowest-order dimension to the highest dimension consecutively, in d

iterations, as follows. During the first iteration, P_0 sends a copy of X to P_1 using the link between P_0 and P_1; during the second iteration, P_0 and P_1 send copies of X to P_2 and P_3, respectively, using the links between P_0 and P_2 and between P_1 and P_3, and so on. The algorithm is stated next.

ALGORITHM 1.6

(Broadcasting from One Processor on the Hypercube)

Input: *Processor P_0 of a $(p = 2^d)$-processor synchronous hypercube holds the data item X in its register $D(0)$.*

Output: *X is broadcast to all the processors such that $D(i) = X$, where $1 \leq i \leq p - 1$.*

Algorithm for Processor P_i

begin
  for $l = 0$ **to** $d - 1$ **do**
    if $0 \leq i \leq 2^l - 1$ **then**
      Set $D(i^{(l)}) := D(i)$
end

Again, as in Algorithm 1.5, the instruction *Set* $D(i^{(l)}) := D(i)$ involves two substeps. In the first substep, a copy of $D(i)$ is sent from processor P_i to processor $P_{i^{(l)}}$ through the existing link between the two processors. In the second substep, $P_{i^{(l)}}$ receives the copy, and stores the copy in its D register.

Clearly, the broadcasting algorithm takes $O(\log p)$ parallel steps. □

The two hypercube algorithms presented belong to the class of **normal algorithms.** The hypercube algorithms in this class use one dimension at each time unit such that consecutive dimensions are used at consecutive time units. Actually, the sum and broadcasting algorithms (Algorithms 1.5 and 1.6) belong to the more specialized class of **fully normal algorithms,** which are normal algorithms with the additional constraint that all the d dimensions of the hypercube are used in sequence (either increasing, as in the broadcasting algorithm, or decreasing, as in the sum algorithm).

EXAMPLE 1.11: (Matrix Multiplication on the Hypercube)

We consider the problem of computing the product of two matrices $C = AB$ on a synchronous hypercube with $p = n^3$ processors, where all matrices are of order $n \times n$.

Let $n = 2^q$ and hence $p = 2^{3q}$. We index the processors by the triples (l, i, j) such that $P_{l,i,j}$ represents processor P_r, where $r = ln^2 + in + j$. In other words, expanding the index r in binary, we obtain that the q most significant bits correspond to the index l, the next q most significant bits correspond to

the index i, and finally the q least significant bits correspond to the index j. In particular, if we fix any pair of indices l, i, and j, and vary the remaining index over all its possible values, we obtain a subcube of dimension q.

The input array A is stored in the subcube determined by the processors $P_{l,i,0}$, where $0 \leq l, i \leq n-1$, such that $A(i, l)$ is stored in processor $P_{l,i,0}$. Similarly, the input array B is stored in the subcube formed by the processors $P_{l,0,j}$ where processor $P_{l,0,j}$ holds the entry $B(l, j)$.

The goal is to compute $C(i,j) = \sum_{l=0}^{n-1} A(i,l)B(l,j)$, for $0 \leq i,j \leq n-1$. The overall algorithm consists of three stages.

1. The input data are distributed such that processor $P_{l,i,j}$ will hold the two entries $A(i, l)$ and $B(l, j)$, for $0 \leq l, i, j \leq n-1$.
2. Processor $P_{l,i,j}$ computes the product $C'(l,i,j) = A(i,l)B(l,j)$, for all $0 \leq i, j, l \leq n-1$.
3. For each $0 \leq i, j \leq n-1$, processors $P_{l,i,j}$, where $0 \leq l \leq n-1$, compute the sum $C(i,j) = \sum_{l=0}^{n-1} C'(l,i,j)$.

The implementation of the first stage consists of two substages. In the first substage, we broadcast, for each i and l, $A(i, l)$ from processor $P_{l,i,0}$ to $P_{l,i,j}$, for $0 \leq j \leq n-1$. Since the set of processors $\{P_{l,i,j} \mid 0 \leq j \leq n-1\}$forms a q-dimensional cube for each pair i and l, we can use the previous broadcasting algorithm (Algorithm 1.6) to broadcast $A(i, l)$ from $P_{l,i,0}$ to all the processors $P_{l,i,j}$. In the second substage, each element $B(l, j)$ held in processor $P_{l,0,j}$ is broadcast to processors $P_{l,i,j}$, for all $0 \leq i \leq n-1$. At the end of the second substage, processor $P_{l,i,j}$ will hold the two entries $A(i, l)$ and $B(l, j)$. Using our broadcasting algorithm (Algorithm 1.6), we can complete the first stage in $2q = O(\log n)$ parallel steps.

The second stage consists of performing one multiplication in each processor $P_{l,i,j}$. Hence, this stage requires one parallel step. At the end of this stage, processor $P_{l,i,j}$ holds $C'(l, i, j)$.

The third stage consists of computing n^2 sums $C(i, j)$; the terms $C'(l, i, j)$ of each sum reside in a q-dimensional hypercube $\{P_{l,i,j} \mid 0 \leq l \leq n-1\}$. As we have seen before (Algorithm 1.5), each such sum can be computed in $q = O(\log n)$ parallel steps. Processor $P_{0,i,j}$ will hold the entry $C(i, j)$ of the product.

Therefore, the product of two $n \times n$ matrices can be computed in $O(\log n)$ time on an n^3-processor hypercube. □

1.3.4 COMPARISON

Although, for a given situation, each of the parallel models introduced could be clearly advantageous, we believe that the shared memory model is most

suited for the general presentation of parallel algorithms. Our choice for the remainder of this book is the PRAM model, a choice justified by the discussion that follows.

In spite of its simplicity, the dag model applies to a specialized class of problems and suffers from several deficiencies. Unless the algorithm is fairly regular, the dag could be quite complicated and very difficult to analyze. The dag model presents only partial information about a parallel algorithm, since a scheduling problem and a processor allocation problem will still have to be resolved. In addition, it has no natural mechanisms to handle communication among the processors or to handle memory allocations and memory accesses.

Although the network model seems to be considerably better suited to resolving both computation and communication issues than is the dag model, its comparison with the shared-memory model is more subtle. For our purposes, the network model has two main drawbacks. First, it is significantly more difficult to describe and analyze algorithms for the network model. Second, the network model depends heavily on the particular topology under consideration. Different topologies may require completely different algorithms to solve the same problem, as we have already seen with the parallel implementation of the standard matrix-multiplication algorithm. These arguments clearly tip the balance in favor of the shared-memory model as a more suitable **algorithmic model.**

The PRAM model, which is the synchronous version of the shared-memory model, draws its power from the following facts:

- There exists a well-developed body of techniques and methods to handle many different classes of computational problems on the PRAM model.
- The PRAM model removes algorithmic details concerning synchronization and communication, and thereby allows the algorithm designer to focus on the structural properties of the problem.
- The PRAM model captures several important parameters of parallel computations. A PRAM algorithm includes an explicit understanding of the operations to be performed at each time unit, and explicit allocation of processors to jobs at each time unit.
- The PRAM design paradigms have turned out to be robust. Many of the network algorithms can be directly derived from PRAM algorithms. In addition, recent research advances have shown that PRAM algorithms can be mapped efficiently on several bounded-degree networks (see the bibliographic notes at the end of this chapter).
- It is possible to incorporate issues such as synchronization and communication into the shared-memory model; hence, PRAM algorithms can be analyzed within this more general framework.

For the remainder of this book, *we use the PRAM model as our formal model to design and analyze parallel algorithms*. Sections 1.4 through 1.6

introduce a convenient framework for presenting and analyzing parallel algorithms. This framework is closely related to what is commonly referred to as *data-parallel* algorithms.

1.4 Performance of Parallel Algorithms

Given a parallel algorithm, we typically measure its performance in terms of worst-case analysis. Let Q be a problem for which we have a PRAM algorithm that runs in time $T(n)$ using $P(n)$ processors, for an instance of size n. The time-processor product $C(n) = T(n) \cdot P(n)$ represents the **cost** of the parallel algorithm. The parallel algorithm can be converted into a sequential algorithm that runs in $O(C(n))$ time. Simply, we have a single processor simulate the $P(n)$ processors in $O(P(n))$ time, for each of the $T(n)$ parallel steps.

The previous argument can be generalized to any number $p \leq P(n)$ of processors as follows. For each of the $T(n)$ parallel steps, we have the p processors simulate the $P(n)$ original processors in $O(P(n)/p)$ substeps: in the first substep, original processors numbered $1, 2, \ldots, p$ are simulated; in the second substep, processors numbered $p + 1, p + 2, \ldots, 2p$ are simulated; and so on. This simulation takes a total of $O(T(n)P(n)/p)$ time.

When the number p of processors is larger than $P(n)$, we can clearly achieve the running time of $T(n)$ by using $P(n)$ processors only.

We have just explained why the following four ways of measuring the performance of parallel algorithms are asymptotically equivalent:

- $P(n)$ processors and $T(n)$ time
- $C(n) = P(n)T(n)$ cost and $T(n)$ time
- $O(T(n)P(n)/p)$ time for any number $p \leq P(n)$ processors
- $O\left(\frac{C(n)}{p} + T(n)\right)$ time for any number p of processors

In the context of the PRAM algorithm to compute the sum of n elements (Example 1.5), these four measures mean (1) n processors and $O(\log n)$ time, (2) $O(n \log n)$ cost and $O(\log n)$ time, (3) $O\left(\frac{n \log n}{p}\right)$ time for any number $p \leq n$ of processors, and (4) $O\left(\frac{n \log n}{p} + \log n\right)$ time for any number p of processors. For the PRAM algorithm to perform matrix multiplication (Example 1.6), the corresponding measures are (1) n^3 processors and $O(\log n)$ time, (2) $O(n^3 \log n)$ cost and $O(\log n)$ time, (3) $O\left(\frac{n^3 \log n}{p}\right)$ time for any number $p \leq n^3$ of processors, and (4) $O\left(\frac{n^3 \log n}{p} + \log n\right)$ time for any number p of processors.

Before we introduce our notion of optimality, we describe the work-time framework for presenting and analyzing parallel algorithms.

1.5 The Work-Time Presentation Framework of Parallel Algorithms

Often, parallel algorithms contain numerous details. Describing parallel algorithms for the PRAM model helps us to simplify the details, because of the relative strength of this model. The description paradigm we shall outline will help us further.

The **work-time (WT) paradigm** provides informal guidelines for a two-level top-down description of parallel algorithms. The upper level suppresses specific details of the algorithm. The lower level follows a general **scheduling principle,** and results in a full PRAM description.

Upper Level (Work-Time (WT) Presentation of Algorithms): Set the following intermediate goal for the design of a parallel algorithm. *Describe the algorithm in terms of a sequence of time units, where each time unit may include any number of concurrent operations.*

We define the **work** performed by a parallel algorithm to be the total number of operations used. Before presenting our next example, we introduce the following **pardo** statement:

for $l \leq i \leq u$ **pardo** statement

The statement (which can be a sequence of statements) following the **pardo** depends on the index i. The statements corresponding to all the values of i between l and u are executed concurrently.

EXAMPLE 1.12:

Consider again the problem of computing the sum S of $n = 2^k$ numbers stored in an array A. Algorithm 1.2, presented in Example 1.5, specifies the program to be executed by each processor P_i, where $1 \leq i \leq n$. The WT presentation of the same algorithm is given next.

ALGORITHM 1.7
(Sum)
Input: $n = 2^k$ numbers stored in an array A.
Output: The sum $S = \sum_{i=1}^{n} A(i)$
begin
1. **for** $1 \leq i \leq n$ **pardo**
   Set $B(i) := A(i)$
2. **for** $h = 1$ **to** $\log n$ **do**
   for $1 \leq i \leq n/2^h$ **pardo**
   Set $B(i) := B(2i - 1) + B(2i)$

3. Set $S: = B(1)$
end

This version of the parallel algorithm contains no mention of how many processors there are, or how the operations will be allocated to processors. It is stated only in terms of time units, where each time unit may include any number of concurrent operations. In particular, we have $\log n + 2$ time units, where n operations are performed within the first time unit (step 1); the jth time unit (iteration $h = j - 1$ of step 2) includes $n/2^{j-1}$ operations, for $2 \leq j \leq \log n + 1$; and only one operation takes place at the last time unit (step 3). Therefore, the work performed by this algorithm is $W(n) = n + \sum_{j=1}^{\log n}(n/2^j) + 1 = O(n)$. The running time is clearly $T(n) = O(\log n)$. □

The main advantage with respect to a PRAM specification is that we do not have to deal with processors. The presence of p processors would have bounded the number of operations to at most p in each unit of time. Furthermore, it would have forced us to allocate each of the processors to execute a specific sequence of operations.

Lower Level: Suppose that the WT presentation of algorithms results in a parallel algorithm that runs in $T(n)$ time units while performing $W(n)$ work (that is, the algorithm requires a total of $W(n)$ operations). Using the general WT scheduling principle given next, we can almost always adapt this algorithm to run on a p-processor PRAM in $\leq \lfloor \frac{W(n)}{p} \rfloor + T(n)$ parallel steps.

The WT Scheduling Principle: Let $W_i(n)$ be the number of operations performed in time unit i, where $1 \leq i \leq T(n)$. Simulate each set of $W_i(n)$ operations in $\leq \lceil \frac{W_i(n)}{p} \rceil$ parallel steps by the p processors, for each $1 \leq i \leq T(n)$ (see Fig. 1.9). If the simulation is successful, the corresponding p-processor PRAM algorithm takes $\leq \sum_i \lceil \frac{W_i(n)}{p} \rceil \leq \sum_i \left(\lfloor \frac{W_i(n)}{p} \rfloor + 1 \right) \leq \lfloor \frac{W(n)}{p} \rfloor + T(n)$ parallel steps, as desired.

A remark concerning the adaptation of the WT scheduling principle is in order. The success of this principle depends on two implementation issues: the first is the calculation of $W_i(n)$ for each i (usually trivial); the second is the allocation of each processor to the appropriate tasks to be performed by that processor. More precisely, for each parallel step, each processor P_k must know whether or not it is active; if it is active, P_k must know the instruction it has to execute and the corresponding operands.

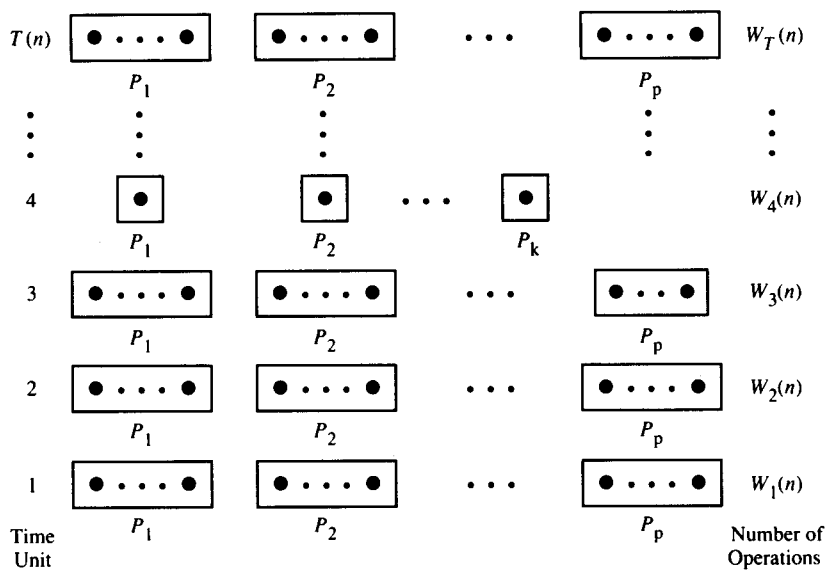


FIGURE 1.9
The WT scheduling principle. During each time unit i, the $W_i(n)$ operations are scheduled as evenly as possible among the available p processors. For example, during time units 1 and 2, each processor is scheduled to execute the same number of operations; during time unit 3, the pth processor executes one less operation than are executed by the remaining processors; and during time unit 4, there are only k possible concurrent operations, which are distributed to the k smallest-indexed processors.

EXAMPLE 1.13:

We now address the implementation details of the WT scheduling principle as related to the PRAM algorithm for computing the sum of n numbers (Algorithm 1.7).

Assume that our PRAM has $p = 2^q \leq n = 2^k$ processors $P_1, P_2, \ldots, P_p$, and let $l = \frac{n}{p} = 2^{k-q}$. The input array A is divided into p subarrays such that processor P_s is responsible for processing the sth subarray $A(l(s-1)+1)$, $A(l(s-1)+2)$, $\ldots$, $A(ls)$. At each height h of the binary tree, the generation of the $B(i)$s is divided in a similar way among the p processors. The number of possible concurrent operations at level h is $n/2^h = 2^{k-h}$. If $2^{k-h} \geq p = 2^q$ (equivalently, $k - h - q \geq 0$, as in Algorithm 1.8, which follows), then these operations are divided equally among the p processors (step 2.1 in Algorithm

1.8). Otherwise ($k - h - q < 0$), the 2^{k-h} lowest-indexed processors execute these operations (step 2.2 in Algorithm 1.8).

The algorithm executed by the sth processor is given next. Figure 1.10 illustrates the corresponding allocation in the case of $n = 8$ and $p = 4$.

ALGORITHM 1.8

(Sum Algorithm for Processor P_s)

Input: *An array A of size $n = 2^k$ stored in the shared memory. The initialized local variables are (1) the order n; (2) the number p of processors, where $p = 2^q \leq n$, and (3) the processor number s.*

Output: *The sum of the elements of A stored in the shared variable S. The array A retains its original value.*

begin

1. **for** $j = 1$ **to** $l\left(= \frac{n}{p}\right)$ **do**
   Set $B(l(s-1)+j) := A(l(s-1)+j)$
2. **for** $h = 1$ **to** $\log n$ **do**

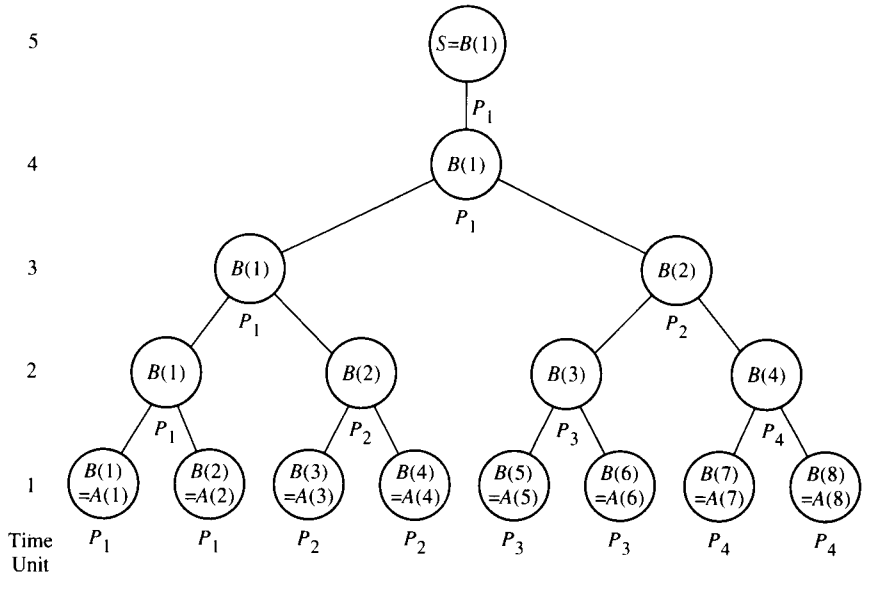


FIGURE 1.10
Processor allocation for computing the sum of eight elements on the PRAM. The operation represented by a node is executed by the processor indicated below the node.

2.1. **if** $(k - h - q \geq 0)$ **then**
for $j = 2^{k-h-q}(s - 1) + 1$ **to** $2^{k-h-q}s$ **do**
Set $B(j)$: $= B(2j - 1) + B(2j)$
2.2. **else** {**if** $(s \leq 2^{k-h})$ **then**
Set $B(s)$: $= B(2s - 1) + B(2s)$}
3. **if** $(s = 1)$ **then** *set* S: $= B(1)$
end

The running time of Algorithm 1.8 can be estimated as follows. Step 1 takes $O\left(\frac{n}{p}\right)$ time, since each processor executes $\frac{n}{p}$ operations. The hth iteration of step 2 takes $O\left(\frac{n}{2^h p}\right)$ time, since a processor has to perform at most $\lceil \frac{n}{2^h p} \rceil$ operations. Step 3 takes $O(1)$ time. Hence, the running time $T_p(n)$ is given by $T_p(n) = O\left(\frac{n}{p} + \sum_{h=1}^{\log n} \lceil \frac{n}{2^h p} \rceil\right) = O\left(\frac{n}{p} + \log n\right)$, as predicted by the WT scheduling principle. □

The prefix-sums algorithm (Algorithm 2.2), presented in the next chapter, will also demonstrate an application of the WT scheduling principle in a slightly more complicated situation. However, as we advance in this book, we describe parallel algorithms for the WT presentation level only. Our justification for omitting the lower-level details is that they usually do not require any new ideas beyond the ones already presented. Typically, they are tedious, cumbersome, programming-type details that would considerably complicate the presentation of the algorithms.

Work Versus Cost: The notion of cost introduced in Section 1.4 and the notion of work introduced in this section are closely related. Given a parallel algorithm running in time $T(n)$ and using a total of $W(n)$ operations (WT presentation level), this algorithm can be simulated on a p-processor PRAM in $T_p(n) = O\left(\frac{W(n)}{p} + T(n)\right)$ time by the WT scheduling principle. The corresponding cost is thus $C_p(n) = T_p(n) \cdot p = O(W(n) + T(n)p)$.

It follows that the two notions coincide (asymptotically) for $p = O\left(\frac{W(n)}{T(n)}\right)$, since we always have $C_p(n) \geq W(n)$, for any p. Otherwise, the two notions differ: $W(n)$ measures the total number of operations used by the algorithm and has nothing to do with the number of processors available, whereas $C_p(n)$ measures the cost of the algorithm relative to the number p of processors available.

Consider, for example, our PRAM algorithm for computing the sum of n numbers (Algorithm 1.7). In this case, we have $W(n) = O(n)$, $T(n) = O(\log n)$, and $C_p(n) = O(n + p \log n)$. When n processors are available, at most $\frac{n}{2}$ processors can be active during the time unit corresponding to the first level of the binary tree, at most $\frac{n}{4}$ processors can be active during the next time unit,

and so on. Therefore, even though our parallel algorithm requires only a total of $O(n)$ operations, the algorithm cannot efficiently utilize the n processors to do useful work.

1.6 The Optimality Notion

Given a computational problem Q, let the sequential time complexity of Q be $T^*(n)$. As mentioned in Section 1.1, this assumption means that there is an algorithm to solve Q whose running time is $O(T^*(n))$; it can be shown that this time bound cannot be improved. A sequential algorithm whose running time is $O(T^*(n))$ is called **time optimal.**

For parallel algorithms, we define two types of optimality; the first is weaker than the second.

A parallel algorithm to solve Q, given in the WT presentation level, will be called **optimal** if the work $W(n)$ required by the algorithm satisfies $W(n) = \Theta(T^*(n))$. In other words, the total of number of operations used by the optimal parallel algorithm is asymptotically the same as the sequential complexity of the problem, *regardless* of the running time $T(n)$ of the parallel algorithm.

We now relate our optimality notion to the speedup notion introduced in Section 1.1. An optimal parallel algorithm whose running time is $T(n)$ can be simulated on a p-processor PRAM in time $T_p(n) = O\left(\frac{T^*(n)}{p} + T(n)\right)$, using the WT scheduling principle. Therefore, the speedup achieved by such an algorithm is given by

$$S_p(n) = \Omega\left(\frac{T^*(n)}{\frac{T^*(n)}{p} + T(n)}\right) = \Omega\left(\frac{pT^*(n)}{T^*(n) + pT(n)}\right).$$

It follows that the algorithm achieves an optimal speedup (that is, $S_p(n) = \Theta(p)$) whenever $p = O\left(\frac{T^*(n)}{T(n)}\right)$. Therefore, the faster the parallel algorithm, the larger the range of p for which the algorithm achieves an optimal speedup.

We have not yet factored the running time $T(n)$ of the parallel algorithm into our notion of optimality. An optimal parallel algorithm is **work-time (WT) optimal** or **optimal in the strong sense** if it can be shown that $T(n)$ cannot be improved by any other *optimal* parallel algorithm. Therefore, the running time of a WT optimal algorithm represents the ultimate speed that can be achieved without sacrificing in the total number of operations.

EXAMPLE 1.14:

Consider the PRAM algorithm to compute the sum given in the WT framework (Algorithm 1.7). We have already noticed that $T(n) = O(\log n)$ and

$W(n) = O(n)$. Since $T^*(n) = n$, this algorithm is optimal. It achieves an optimal speedup whenever the number p of processors satisfies $p = O(\frac{n}{\log n})$. On the other hand, we shall see in Chapter 10 that any CREW PRAM will require $\Omega(\log n)$ time to compute the sum, regardless of the number of processors available. Therefore, this algorithm is WT optimal for the CREW PRAM. □

1.7 *Communication Complexity

The communication complexity of a PRAM algorithm, defined as *the worst-case bound on the traffic between the shared memory and any local memory of a processor*, is an important factor to consider in estimating the actual performance of these algorithms. We shed light on this issue as it relates to our parallel-algorithms framework.

Given a high-level description of a parallel algorithm A, a successful adaptation of the WT scheduling principle for p processors will always yield a parallel algorithm with the best possible computation time relative to the running time of A. However, the communication complexity of the corresponding adaptation is not necessarily the best possible. We illustrate this point by revisiting the matrix-multiplication problem (Example 1.6), which is stated next in the WT presentation framework.

ALGORITHM 1.9
(Matrix Multiplication Revisited)
Input: *Two $n \times n$ matrices A and B stored in the shared memory, where $n = 2^k$ for some integer k.*
Output: *The product $C = AB$ stored in the shared memory.*
begin
1. **for** $1 \le i, j, k \le n$ **pardo**
   Set $C'(i, j, l) := A(i, l)B(l, j)$
2. **for** $h = 1$ **to** $\log n$ **do**
   for $1 \le i, j \le n, 1 \le l \le n/2^h$ **pardo**
   Set $C'(i, j, l) := C'(i, j, 2l - 1) + C'(i, j, 2l)$
3. **for** $1 \le i, j \le n$ **pardo**
   Set $C(i, j) := C'(i, j, 1)$

end

Algorithm 1.9 runs in $O(\log n)$ time using a total of $O(n^3)$ operations. By the WT scheduling principle, the algorithm can be simulated by p processors to run in $O(n^3/p + \log n)$ time.

Consider the adaptation of this algorithm to the case where there are n processors available. In particular, the corresponding running time must be $O(n^2)$. We examine the communication complexity of Algorithm 1.9 relative to a particular processor allocation scheme.

A straightforward scheme proceeds by allocating the operations included in each time unit to the available processors (as in the statement of the WT scheduling principle). In particular, the n^3 concurrent operations of step 1 can be allocated equally among the n processors as follows. For each $1 \leq i \leq n$, processor P_i computes $C'(i,j,l) = A(i,l)B(l,j)$, where $1 \leq j,l \leq n$; hence, P_i has to read the ith row of A and all of matrix B from the shared memory. A traffic of $O(n^2)$ numbers is created between the shared memory and each of the local memories of the processors.

The hth iteration of the loop at step 2 requires $n^3/2^h$ concurrent operations, which can be allocated as follows. Processor P_i's task is to update the values $C'(i,j,l)$, for all $1 \leq j,l \leq n$; hence, P_i can read all the necessary values $C'(i,j,l)$ for all indices $1 \leq j,l \leq n$, and can then perform the operations required on this set of values. Again, $O(n^2)$ entries get swapped between the shared memory and the local memory of each processor.

Finally, step 3 can be implemented easily with $O(n)$ communication, since processor P_i has to store the ith row of the product matrix in the shared memory, for $1 \leq i \leq n$.

Therefore, we have a processor allocation scheme that adapts the *WT* scheduling principle successfully and that has a communication requirement of $O(n^2)$.

We now develop another parallel implementation of the standard matrix-multiplication algorithm that will result in many fewer data elements being transferred between the shared memory and each of the local memories of the n processors. In addition, the computation time remains the same.

Assume, without loss of generality, that $\alpha = \sqrt[3]{n}$ is an integer. Partition matrix A into $\sqrt[3]{n} \times \sqrt[3]{n}$ blocks of submatrices, each of size $n^{2/3} \times n^{2/3}$, as follows:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,\alpha} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,\alpha} \\ \cdots & \cdots & \cdots & \cdots \\ A_{\alpha,1} & A_{\alpha,2} & \cdots & A_{\alpha,\alpha} \end{bmatrix}.$$

We partition B and C in the same way. Notice that there are exactly n pairs $(A_{i,l}, B_{l,j})$ for all i, j, and l.

The new processor allocation follows a strategy different from the one outlined in the WT scheduling principle. Each processor P reads a unique pair $(A_{i,l}, B_{l,j})$ of blocks from A and B, respectively, and computes the product $D_{i,j,l} = A_{i,l}B_{l,j}$, which is then stored in the shared memory. The amount of communication needed is $O(n^{4/3})$, which accounts for the cost of transferring a pair of blocks from the shared memory into each local memory

and the cost of storing a new block from each local memory into the shared memory. On the other hand, the amount of computation required for performing $D_{i,j,l} = A_{i,l}B_{l,j}$ is $O(n^2)$, since each block is of size $n^{2/3} \times n^{2/3}$.

Next, each block $C_{i,j}$ of the product matrix C is given by $C_{i,j} = \sum_{l=1}^{\sqrt[3]{n}} D_{i,j,l}$, and there are $n^{2/3}$ such blocks. We can now allocate $\sqrt[3]{n}$ processors to compute each block $C_{i,j}$ such that the computation proceeds in the fashion of a balanced binary tree whose $\sqrt[3]{n}$ leaves contain the blocks $D_{i,j,l}$, where $1 \leq l \leq \sqrt[3]{n}$. Each level of the tree requires the concurrent access of a set of blocks each of size $n^{2/3} \times n^{2/3}$. Hence, the execution of the operations represented by each level of a tree requires $O(n^{4/3})$ communication. Therefore the total amount of communication required for computing all the C_{ij}'s is $O(n^{4/3} \log n)$, which is substantially smaller than the $O(n^2)$ required by the previous processor allocation scheme.

Remark 1.3: We can reduce to $O(n^{4/3})$ the communication cost of the second processor allocation scheme to implement the standard matrix-multiplication algorithm by using the pipelining technique introduced in Chapter 2. □

In summary, a parallel algorithm given in the WT presentation level requires a solution to a processor allocation problem that will uniquely determine the computation and the communication requirements of the algorithm. In almost all cases, the strategy outlined in the statement of the WT scheduling principle results in an optimal allocation of the computation among the available processors. However, the communication complexity of the resulting implementation may not be optimal. An alternative parallel algorithm may be required to achieve optimal communication complexity.

1.8 Summary

The design and analysis of parallel algorithms involve a complex set of interrelated issues that is difficult to model appropriately. These issues include computational concurrency, processor allocation and scheduling, communication, synchronization, and granularity (granularity is a measure of the amount of computation that can be performed by the processors between synchronization points). An attempt to capture most of the related parameters makes the process of designing parallel algorithms a challenging task. We have opted for simplicity and elegance, while attempting to shed light on some of the important performance issues arising in the design of parallel algorithms.

In this chapter, we introduced two major parallel-computation models: the shared memory and the network. These models offer two orthogonal approaches to parallel computation—shared-memory computation versus distributed-memory computation. We argued in favor of the shared-memory model due to its simplicity, uniformity, and elegance.

We described a high-level framework for presenting and analyzing PRAM algorithms. We identified the two important parameters, time and work. *Time* refers to the number of time units required by the algorithm, where during each time unit a number of concurrent operations can take place. *Work* refers to the total number of operations needed to execute the algorithm. A scheduling principle translates such an algorithm into a parallel algorithm running on a PRAM with any number of processors.

Our notion of optimality concentrates on minimizing the total work primarily and the running time secondarily. A more detailed analysis could also incorporate the communication requirement.

The WT presentation framework is used almost exclusively in the remainder of this book. It allows clear and succinct presentation of fairly complicated parallel algorithms. It is also closely related to data-parallel programming, which has been used for shared and distributed-memory computations, as well as to SIMD and MIMD parallel computers.

Exercises

1.1. We have seen how to schedule the dag corresponding to the standard algorithm for multiplying two $n \times n$ matrices in $O(\log n)$ time using n^3 processors. What is the optimal schedule for an arbitary number p of processors, where $1 \leq p \leq n^3$? What is the corresponding parallel complexity?

1.2. **a.** Consider the problem of computing X^n, where $n = 2^k$ for some integer k. The *repeated-squaring* algorithm consists of computing $X^2 = X \times X, X^4 = X^2 \times X^2, X^8 = X^4 \times X^4$, and so on. Draw the dag corresponding to this algorithm. What is the optimal schedule for p processors, where $1 \leq p \leq n$?

b. Draw the dag and give the optimal schedule for the case when X is an $m \times m$ matrix?

1.3. Let A be an $n \times n$ lower triangular matrix such that $a_{ii} \neq 0$, for $1 \leq i \leq n$, and let b be an n-dimensional vector. The *back-substitution* method to solve the linear system of equations $Ax = b$ begins by determining x_1 using the first equation ($a_{11}x_1 = b_1$), then determining x_2 using the second equation ($a_{21}x_1 + a_{22}x_2 = b_2$), and so on.

Let G be the dag corresponding to the back-substitution algorithm.

a. Determine an optimal schedule of G for any number $p \leq n$ of processors. What is the corresponding speedup?

b. Determine an optimal schedule where $p > n$. What is the best possible speedup achievable in this case?

1.4. Let $p(x) = a_0x^n + a_1x^{n-1} + \cdots + a_{n-1}x + a_n$ be a given polynomial. *Horner's algorithm* to compute $p(x)$ at a point x_0 is based on rewriting the expression for $p(x_0)$ as follows:

$$p(x_0) = (\cdots((a_0x_0 + a_1)x_0 + a_2)x_0 + \cdots + a_{n-1})x_0 + a_n.$$

a. Draw the dag corresponding to Horner's algorithm. What is the optimal schedule for p processors, $1 \leq p \leq n$?

b. Is it possible to develop a parallel algorithm whose complexity is $O\left(\frac{n}{p} + \log n\right)$ for p processors?

1.5. **a.** The global memory of an EREW PRAM contains n bits x_i stored in n consecutive locations. Given p processors, where $p \leq n$, develop the algorithm that has to be executed by each processor to compute the Boolean AND of the x_i's. Your algorithm must run in $O\left(\frac{n}{p} + \log n\right)$ time.

b. Show how to perform the same computation in $O\left(\frac{n}{p}\right)$ time on the common CRCW PRAM.

1.6. Rewrite the PRAM algorithm to multiply two $n \times n$ matrices (Algorithm 1.3) for the case when the processors are indexed from 1 to n^3 (instead of the indexing (i, j, l), for $1 \leq i, j, l \leq n$).

1.7. Our PRAM algorithm to multiply two $n \times n$ matrices (Algorithm 1.3) assumed the presence of n^3 processors. Assume that you have only p processors, where $1 \leq p \leq n^3$. Develop the corresponding algorithm to be executed by an arbitrary processor P_r, where $1 \leq r \leq p$. What is the running time of your algorithm? What is the corresponding speedup relative to the standard matrix-multiplication algorithm? Assume that $n = 2^k$, and $p = 2^q$.

1.8. An item X is stored in a specified location of the global memory of an EREW PRAM. Show how to broadcast X to all the local memories of the p processors of the EREW PRAM in $O(\log p)$ time. Determine how much time it takes to perform the same operation on the CREW PRAM.

1.9. Design a systolic algorithm to compute the matrix-vector product Ab, where A is an $n \times n$ matrix and b is an n-dimensional vector, on a linear array of n processors. Your algorithm must run in $O(n)$ time.

1.10. Suppose two $n \times n$ matrices A and B are initially stored on a mesh of n^2 processors such that P_{ij} holds $A(i, j)$ and $B(j, i)$. Develop an asynchronous algorithm to compute the product of the two matrices in $O(n)$ time.

1.11. Develop an algorithm to multiply two $n \times n$ matrices on a mesh with p^2 processors, where $1 \leq p < n$. You can assume that the two matrices are initially stored in the mesh in any order you wish, as long as each processor has the same number of entries. What is the corresponding speedup? Assume that p divides n.

1.12. Assume that processor P_{ij} of an $n \times n$ mesh has an element X to be broadcast to all the processors. Develop an asynchronous algorithm to perform the broadcasting. What is the running time of your algorithm?

1.13. Given a sequence of numbers $x_1, x_2, \cdots, x_n$, the *prefix sums* are the partial sums $s_1 = x_1, s_2 = x_1 + x_2, \cdots, s_n = \sum_{i=1}^{n} x_i$. Show how to compute the prefix sums on a mesh with n processors in $O(\sqrt{n})$ time. What is the corresponding speedup? Assume that $\sqrt{n}$ is an integer.

1.14. Assume that an $n \times n$ matrix A is stored on an $n \times n$ mesh of processors such that processor P_{ij} holds the entry $A(i, j)$. Develop an $O(n)$ time algorithm to transpose the matrix such that processor P_{ij} will hold the entry $A(j, i)$. Compare your algorithm with that for computing the transpose of a matrix on the PRAM model.

1.15. Show that any algorithm to compute the transpose of an $n \times n$ matrix (see Exercise 1.14) on a $p \times p$ mesh requires $\Omega\left(\frac{n^2}{p} + p\right)$ time, where $2 \leq p \leq n$. The memory mapping of an $n \times n$ matrix A into a $p \times p$ mesh can be defined as follows. Let p divide n evenly. Partition A into $p \times p$ blocks $\{A_{ij}\}$, each containing n^2/p^2 elements. Processor P_{ij} of the mesh holds the block A_{ij}.

1.16. Let P_i and P_j be two processors of a d-dimensional hypercube. Show that there exist d node-disjoint paths between P_i and P_j such that the length of each path is at most $H(i, j) + 2$, where $H(i, j)$ is the number of bit positions on which i and j differ.

1.17. An embedding of a graph $G = (V, E)$ into a d-dimensional hypercube is a one-to-one mapping f from V into the nodes of the hypercube such that $(i, j) \in E$ implies that processors $P_{f(i)}$ and $P_{f(j)}$ are adjacent in the cube.

a. Develop an embedding of a linear processor array with $p = 2^d$ processors into a d-dimensional hypercube.

b. Develop an embedding of a mesh into a hypercube.

1.18. Develop an $O(n)$ time algorithm to compute the product of two $n \times n$ matrices on the hypercube with n^2 processors.

1.19. Consider the problem of multiplying two $n \times n$ matrices on a synchronous hypercube with $p = n^3$ processors, where $n = 2^q$. Assume that the input arrays A and B are initially stored in processors $P_0, P_1, \cdots, P_{n^2-1}$, where P_k holds the entries $A(i, j)$ and $B(i, j)$ such that $k = in + j$, for

$0 \leq i,j \leq n-1$. Develop a parallel algorithm to compute the product AB in $O(\log n)$ time.

1.20. Our hypercube algorithm to compute the sum of n numbers assumes the availability of n processors. Design an algorithm where we have an arbitrary number p of processors, $1 \leq p \leq n$. What is the corresponding speedup? Is it always optimal?

1.21. Given a sequence of $n = 2^d$ elements $(x_0, x_1, \ldots, x_{n-1})$ stored on a d-dimensional hypercube such that x_i is stored in P_i, where $0 \leq i \leq n-1$, develop an $O(\log n)$ time algorithm to determine the number of x_i's that are smaller than a specified element initially stored in P_0.

1.22. **a.** Develop an algorithm to compute the prefix sums of n elements, as introduced in Exercise 1.13, on a hypercube with $n = 2^d$ processors. Your algorithm should run in $O(\log^2 n)$ time (or faster).

b. Develop an algorithm where the number p of processors is smaller than the number n of elements and each processor holds initially $\frac{n}{p}$ elements. What is the corresponding running time?

1.23. The p processors of a d-dimensional hypercube hold n items such that each processor has at most M items. Develop an algorithm to redistribute the n items such that each processor has exactly $\frac{n}{p}$ items (assuming that p divides n evenly). State the running time of your algorithm as a function of p, M, and n.

1.24. The WT scheduling principle was discussed in the context of PRAM algorithms. Prove that this principle will always work in the dag model.

1.25. Compare a parallel algorithm given in the WT presentation framework to a dag with a given schedule. In particular, determine whether such an algorithm can always be represented by a dag whose optimal schedule meets the time steps of the algorithm.

1.26. **a.** Determine the communication complexity of the PRAM sum algorithm (Algorithm 1.8) as a function of n and p. Is it possible to do better?

b. In general, suppose we are given a linear work, $T(n)$-time PRAM algorithm to solve a problem P of size n. Show that a successful adaptation of the scheduling principle results in an optimal communication complexity as long as $p = O(n/T(n))$.

1.27. Consider the problem of multiplying two $n \times n$ matrices on a PRAM model where each processor has a local memory of size $M \leq n$. Compare the communication complexities of the two parallel implementations of the standard matrix-multiplication algorithm discussed in Section 1.7.

1.28. Generalize the processor allocation scheme described for the $n \times n$ matrix multiplication problem on the PRAM whose communication cost is $O(n^{4/3} \log n)$ to the case when there are p processors available, $1 \leq p \leq n^2$. What is the resulting communication complexity?

Bibliographic Notes

The three parallel models introduced in this chapter have received considerable attention in the literature. Dags have been widely used to model algorithms especially for numerical computations (an early reference is [6]). More advanced parallel algorithms for this model and a discussion of related issues can be found in [5]. Some of the early algorithms for shared-memory models have appeared in [4, 9, 10, 16, 17, 19, 24, 26]. Rigorous descriptions of shared-memory models were introduced later in [11, 12]. The WT scheduling principle is derived from a theorem in [7]. In the literature, this principle is commonly referred to as *Brent's theorem* or *Brent's scheduling principle*. The relevance of this theorem to the design of PRAM algorithms was initially pointed out in [28]. The mesh is perhaps one of the earliest parallel models studied in some detail. Since then, many networks have been proposed for parallel processing. The recent books [2, 21, 23] give more advanced parallel algorithms on various networks and additional references. Recent work on the mapping of PRAM algorithms on bounded-degree networks is described in [3, 13, 14, 20, 25]. Our presentation on the communication complexity of the matrix-multiplication problem in the shared-memory model is taken from [1]. Data-parallel algorithms are described in [15].

Parallel architectures have been described in several books (see, for example, [18, 29]). The necessary background material for this book can be found in many textbooks, including [8, 22, 27].

References

1. Aggarwal, A., A. K. Chandra, and M. Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71(1):3–28, 1990.
2. Akl, S. G. *The Design and Analysis of Parallel Algorithms.* Prentice-Hall, Englewood Cliffs, NJ, 1989.
3. Alt., H., T. Hagerup, K. Mehlhorn, and F. P. Preparata. Simulation of idealized parallel computers on more realistic ones. *SIAM J. Computing*, 16(5):808–835, 1987.
4. Arjomandi, E. *A Study of Parallelism in Graph Theory.* PhD thesis, Computer Science Department, University of Toronto, Toronto, Canada, 1975.
5. Bertsekas, D. P., and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice-Hall, Englewood Cliffs, NJ, 1989.
6. Borodin, A., and I. Munro. *The Computational Complexity of Algebraic and Numeric Problems.* American Elsevier, New York, 1975.
7. Brent, R. P. The parallel evaluation of general arithmetic expressions. *JACM*, 21(2):201–208, 1974.
8. Cormen, T. H., C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, and McGraw-Hill, New York, 1990.
9. Csanky, L. Fast parallel matrix inversion algorithms. *SIAM J. Computing*, 5(4):618–623, 1976.
10. Eckstein, D. M. *Parallel Processing Using Depth-First Search and Breadth-First Search.* PhD thesis, Computer Science Department, University of Iowa, Iowa City, IA, 1977.

11. Fortune, S., and J. Wyllie. Parallelism in random access machines. In *Proceedings Tenth Annual ACM Symposium on Theory of Computing*, San Diego, CA, 1978, pages 114–118. ACM Press, New York.
12. Goldschlager, L. M. A unified approach to models of synchronous parallel machines. In *Proceedings Tenth Annual ACM Symposium on Theory of Computing*, San Diego, CA, 1978, pages 89–94. ACM Press, New York.
13. Herley, K. T. Efficient simulations of small shared memories on bounded degree networks. In *Proceedings Thirtieth Annual Symposium on Foundations of Computer Science*, Research Triangle Park, NC, 1989, pages 390–395. IEEE Computer Society Press, Los Alamitos, CA.
14. Herley, K. T., and G. Bilardi. Deterministic simulations of PRAMs on bounded-degree networks. In *Proceedings Twenty-Sixth Annual Allerton Conference on Communication, Control and Computation*, Monticello, IL, 1988, pages 1084–1093.
15. Hillis, W. D., and G. L. Steele. Data parallel algorithms. *Communication of the ACM*, 29(12):1170–1183, 1986.
16. Hirschberg, D. S. Parallel algorithms for the transitive closure and the connected components problems. In *Proceedings Eighth Annual ACM Symposium on Theory of Computing*, Hershey, PA, 1976, pages 55–57. ACM Press, New York.
17. Hirschberg, D. S. Fast parallel sorting algorithms. *Communication of the ACM*, 21(8):657–661, 1978.
18. Hwang, K., and F. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, New York, 1984.
19. JáJá, J. Graph connectivity problems on parallel computers. Technical Report CS-78-05, Pennsylvania State University, University Park, PA, 1978.
20. Karlin, A., and E. Upfal. Parallel hashing—an efficient implementation of shared memory. *SIAM J. Computing*, 35(4):876–892, 1988.
21. Leighton, T. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, and Hypercubes.* Morgan Kaufmann, San Mateo, CA, 1991.
22. Manber, U. *Introduction to Algorithms: A Creative Approach.* Addison-Wesley, Reading, MA, 1989.
23. Miller, R., and Q. F. Stout. *Parallel Algorithms for Regular Architectures.* MIT Press, Cambridge, MA, 1992.
24. Preparata, F. P. New parallel sorting schemes. *IEEE Transactions Computer*, C-27(7):669–673, 1978.
25. Ranade, A. G. How to emulate shared memory. In *Proceedings Twenty-Eighth Annual Symposium on the Foundations of Computer Science*, Los Angeles, CA, 1987, pages 185–192. IEEE Press, Piscataway, NJ.
26. Savage, C. *Parallel Algorithms for Graph Theoretic Problems.* PhD thesis, Computer Science Department, University of Illinois, Urbana, IL, 1978.
27. Sedgewick, R. *Algorithms.* Addison-Wesley, Reading, MA, 1983.
28. Shiloach, Y. and U. Vishkin. An $O(n^2 \log n)$ parallel max-flow algorithm. *Journal of Algorithms*, 3(2):128–146, 1982.
29. Stone, H. S. *High-Performance Computer Architecture.* Addison-Wesley, Reading, MA, 1987.

2

Basic Techniques

The task of designing parallel algorithms presents challenges that are considerably more difficult than those encountered in the sequential domain. The lack of a well-defined methodology is compensated by a collection of techniques and paradigms that have been found effective in handling a wide range of problems. This chapter introduces these techniques as they apply to a selected set of combinatorial problems, which are interesting on their own and often appear as subproblems in numerous computations.

It is important to view the introduced techniques as general guidelines for designing parallel algorithms, rather than as a manual of directly applicable methods. The combinatorial problems discussed include the prefix sums (Section 2.1), parallel prefix (Section 2.2), the convex hull (Section 2.3), merging (Section 2.4), insertions into 2-3 trees (Section 2.5), computing the maximum (Section 2.6), and coloring the vertices of a directed cycle (Section 2.7). Most of these parallel algorithms are used frequently in the remainder of this book.

2.1 Balanced Trees

The PRAM algorithm to compute the sum of n elements presented in Section 1.3.2 is based on a balanced binary tree whose leaves are the given n elements

and whose internal nodes represent additions. This algorithm is an example of the general strategy of *building a balanced binary tree on the input elements and traversing the tree forward and backward to and from the root*. An internal node u usually holds information concerning the data stored at the leaves of the subtree rooted at u. The success of such a strategy depends partly on the existence of a fast method to determine the information stored in an internal node from the information stored in its children. We use the computation of the prefix sums of n elements to provide another illustration of this strategy.

2.1.1 AN OPTIMAL PREFIX-SUMS ALGORITHM

Consider a sequence of n elements $\{x_1, x_2, \ldots, x_n\}$ drawn from a set S with a binary **associative** operation, denoted by $*$. The **prefix sums** of this sequence are the n partial sums (or products) defined by

$$s_i = x_1 * x_2 * \ldots * x_i, \ 1 \leq i \leq n.$$

A trivial sequential algorithm computes s_i from s_{i-1} with a single operation by using the identity $s_i = s_{i-1} * x_i$, for $2 \leq i \leq n$, and hence takes $O(n)$ time. Clearly, this algorithm is inherently sequential.

We can use a balanced binary tree to derive a fast parallel algorithm to compute the prefix sums. Each internal node represents the application of the operation $*$ to its children during a forward traversal of the tree. Hence, each node v holds the sum of the elements stored in the leaves of the subtree rooted at v. During a backward traversal of the tree, the prefix sums of the data stored in the nodes at a given height are computed.

We start by giving a recursive version described by the steps needed to obtain the data stored in the nodes at height 1 and the steps needed to obtain the prefix sums after the recursive call on the nodes at height 1 terminates.

ALGORITHM 2.1
(Prefix Sums)
Input: *An array of* $n = 2^k$ *elements* $(x_1, x_2, \ldots, x_n)$, *where* k *is a nonnegative integer.*
Output: *The prefix sums* s_i, *for* $1 \leq i \leq n$.
begin
1. **if** $n = 1$ **then** {*set* $s_1 := x_1$; **exit**}
2. **for** $1 \leq i \leq n/2$ **pardo**
   Set $y_i := x_{2i-1} * x_{2i}$
3. *Recursively, compute the prefix sums of* $\{y_1, y_2, \ldots, y_{n/2}\}$, *and store them in* $z_1, z_2, \ldots, z_{n/2}$.

4. **for** $1 \leq i \leq n$ **pardo**
   $\{i$ *even* $\quad$: *set* s_i: $= z_{i/2}$
   $i = 1 \quad$: *set* s_1: $= x_1$
   i *odd* $> 1 \quad$: *set* s_i: $= z_{(i-1)/2} * x_i\}$

end

EXAMPLE 2.1:

The prefix-sums algorithm on eight elements is illustrated in Fig. 2.1. The time units referred to in what follows are those indicated in the figure. During the first time unit, the four elements $y_1 = x_1 * x_2, y_2 = x_3 * x_4, y_3 = x_5 * x_6$, and $y_4 = x_7 * x_8$ are computed. The second time unit corresponds to computing $y_1' = y_1 * y_2$ and $y_2' = y_3 * y_4$ with a recursive call to handle these two inputs. The element $y_1'' = y_1' * y_2'$ is computed during time unit 3. Hence, at the fourth time unit, the prefix sum of this single input is generated. The reverse process begins by generating, in time unit 5, the prefix sums z_1' and z_2' of the two inputs y_1' and y_2' generated during the second time unit. Similarly, during time unit 6, the prefix sums z_1, z_2, z_3, and z_4 of the four elements y_1, y_2, y_3, and y_4 are generated. Finally, the prefix sums $\{s_i\}$ of the x_i's are generated in time unit 7. □

On inputs of size $n = 2^k$, the prefix-sums algorithm requires $2k + 1$ time units such that, during the first k time units, the computation advances from the leaves of a complete binary tree to the root, where the leaves hold x_1, $x_2, \ldots, x_n$. During the last k time units, we transverse the tree in reverse order and compute prefix sums by using the data generated during the first k time units.

We note that the whole algorithm can be executed in place, and that we introduced additional variables for clarity.

We are now ready to examine the following theorem, which is stated within the WT presentation level.

Theorem 2.1: *The Prefix-Sums Algorithm (Algorithm 2.1) computes the prefix sums of n elements in time* $T(n) = O(\log n)$, *using a total of* $W(n) = O(n)$ *operations.*

Proof: The correctness proof will be by induction on k, where the size of the input is $n = 2^k$.

The base case $k = 0$ is handled correctly by step 1 of the algorithm.

Assume that the algorithm works correctly for all sequences of length $n = 2^k$, where $k > 0$. We will prove that the algorithm computes the prefix sums of any sequence of length $n = 2^{k+1}$.

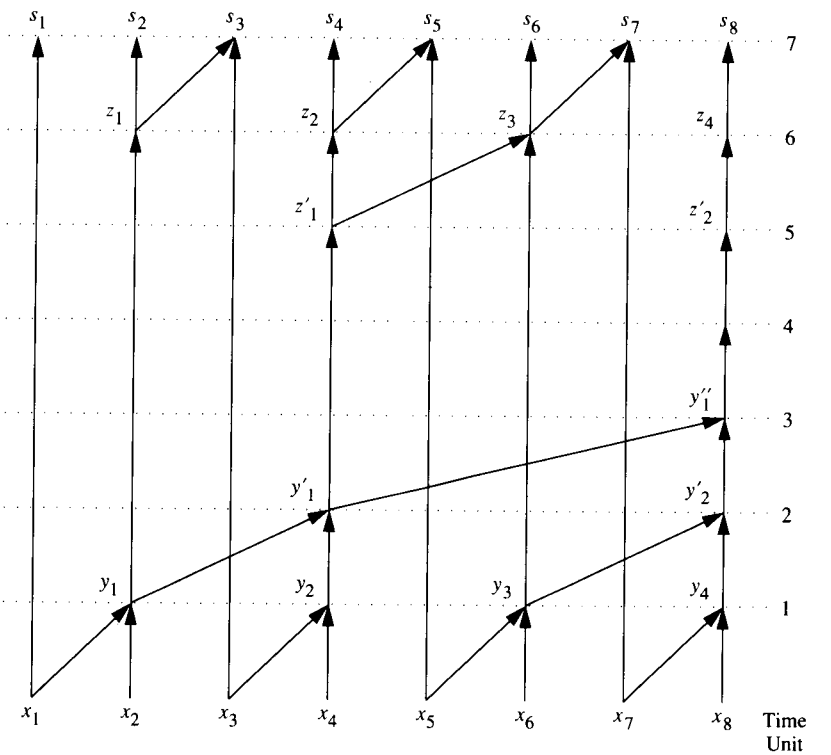


FIGURE 2.1
Prefix sums of eight elements. Each node represents a $*$ operation on the data stored in the tails of the incident arcs. During each time unit, the operations that take place are indicated by the presence of nodes with incoming arrows.

By the induction hypothesis, the variables $z_1, z_2, \ldots, z_{n/2}$, computed at step 3, hold the prefix sums of the sequence $\{y_1, y_2, \ldots, y_{n/2}\}$, where $y_i = x_{2i-1} * x_{2i}$, for $1 \leq i \leq n/2$. In particular, $z_j = y_1 * y_2 * \cdots * y_j$ and hence $z_j = x_1 * x_2 * \cdots * x_{2j-1} * x_{2j}$. That is, each z_j is nothing but the prefix sum s_{2j}, for $1 \leq j \leq n/2$. Thus, if i is even—say, $i = 2j$—then we have that $s_i = z_{i/2}$; otherwise, either $i = 1$ or $i = 2j + 1$, for some $1 \leq j \leq (n/2) - 1$. The case when $i = 1$ is trivial. For the remaining case of $i = 2j + 1$, we have $s_i = s_{2j+1} = s_{2j} * x_{2j+1} = z_{(i-1)/2} * x_i$. Therefore, all these cases are handled appropriately in step 4 of the algorithm. It follows that the algorithm works correctly on all inputs.

As for the resources required, they can be estimated as follows. Step 1 takes $O(1)$ sequential time. Steps 2 and 4 can be executed in $O(1)$ parallel

steps using $O(n)$ operations. Therefore, the running time $T(n)$ and the work $W(n)$ required by the algorithm satisfy the following recurrences:

$$T(n) = T\left(\frac{n}{2}\right) + a$$

$$W(n) = W\left(\frac{n}{2}\right) + bn,$$

where a and b are constants. The solutions of these recurrences are $T(n) = O(\log n)$ and $W(n) = O(n)$. □

PRAM Model: Since steps 1, 2 and 4 of Algorithm 2.1 do not require concurrent read or concurrent write capability, this algorithm runs on the EREW PRAM model. A lower bound of $\Omega(\log n)$ on the time it takes a CREW PRAM to compute the Boolean OR of n variables (regardless of the number of operations used) implies that the previous algorithm is *WT optimal*, or *optimal in the strong sense, on the EREW and CREW PRAM*. This lower-bound proof is presented in Chapter 10. □

2.1.2 A NONRECURSIVE PREFIX-SUMS ALGORITHM

We present a nonrecursive version of the prefix algorithm, which will be used in Section 2.1.3 to illustrate the details involved in adapting the WT scheduling principle.

Let $A(i) = x_i$, where $1 \le i \le n$. Let $B(h,j)$ and $C(h,j)$ be sets of auxiliary variables, where $0 \le h \le \log n$ and $1 \le j \le n/2^h$. The array B will be used to record the information in the binary tree nodes during a forward traversal, whereas the array C will be used during the backward traversal of the tree. Figure 2.2 illustrates the forward traversal, and Fig. 2.3 illustrates the backward traversal, for $n = 8$.

ALGORITHM 2.2
(Nonrecursive Prefix Sums)
Input: *An array A of size $n = 2^k$, where k is a nonnegative integer.*
Output: *An array C such that $C(0, j)$ is the jth prefix sum, for $1 \le j \le n$.*
begin
1. **for** $1 \le j \le n$ **pardo**
   Set $B(0,j) := A(j)$
2. **for** $h = 1$ **to** $\log n$ **do**
   for $1 \le j \le n/2^h$ **pardo**
   Set $B(h,j) := B(h-1, 2j-1) * B(h-1, 2j)$

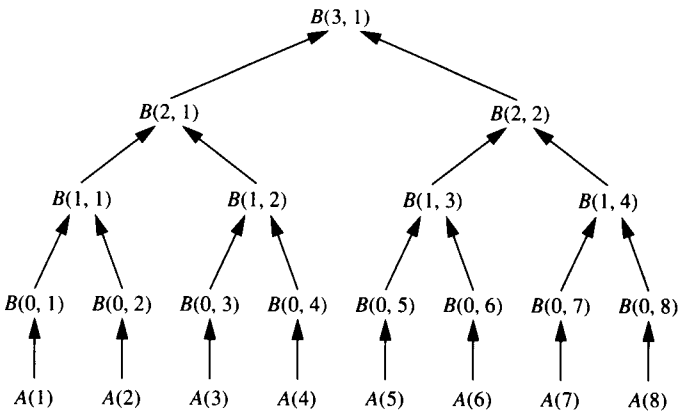


FIGURE 2.2
The bottom-up (forward) traversal of the binary tree used in the nonrecursive prefix sums algorithm. $B(0, j)$ is initially set to $A(j)$, and each internal node represents the operation $*$.

3. **for** $h = \log n$ **to** 0 **do**
   for $1 \le j \le n/2^h$ **pardo**
   $\{j$ *even* : *Set* $C(h, j) := C\left(h + 1, \frac{j}{2}\right)$
   $j = 1$: *Set* $C(h, 1) := B(h, 1)$
   j *odd* > 1 : *Set* $C(h, j) := C\left(h + 1, \frac{j-1}{2}\right) * B(h, j)\}$

end

EXAMPLE 2.2:

Let $n = 8$ (see Figs. 2.2 and 2.3). We initially set $B(0, j) = A(j)$, for all $1 \le j \le 8$. The $B(0, j)$s correspond to the leaves of the binary tree. The variables $B(1, j)$, where $1 \le j \le 4$, correspond to the internal nodes at height 1; the variables $B(2, j)$, where $1 \le j \le 2$, correspond to the internal vertices at height 2. The root of the binary tree is stored in $B(3, 1)$. We then traverse this binary tree backward. We start by setting $C(3, 1) = B(3, 1)$. At the next step, we generate $C(2, 1) = B(2, 1)$ and $C(2, 2) = B(3, 1)$. Hence, $C(2, 1)$ and $C(2, 2)$ hold the prefix sums corresponding to the inputs $B(2, 1)$ and $B(2, 2)$. Similarly, $C(1, 1)$, $C(1, 2)$, $C(1, 3)$, and $C(1, 4)$ are the prefix sums of $B(1, 1)$, $B(1, 2)$, $B(1, 3)$, and $B(1, 4)$. Therefore the $C(0, j)$s hold the prefix sums of the original inputs. □

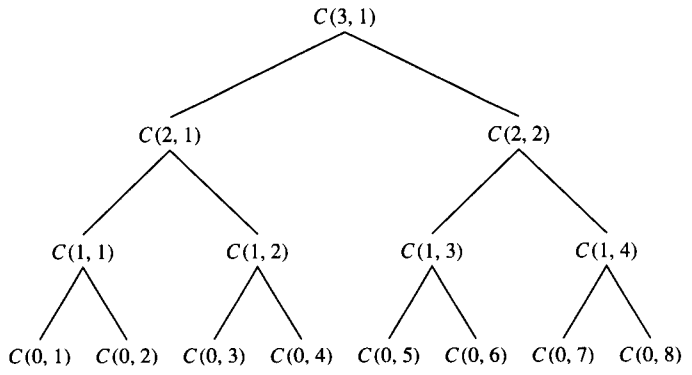


FIGURE 2.3
The elements of the array C as generated by the top-down (backward) traversal of the binary tree corresponding to the nonrecursive prefix-sums algorithm.

2.1.3 *ADAPTATION OF THE WT SCHEDULING PRINCIPLE

As indicated in Section 1.5, the adaptation of the WT scheduling principle requires the determination of the number of operations at each parallel step and a solution to the corresponding processor allocation problem.

Let $n = 2^k$. There are $2 \log n + 2 = 2k + 2$ parallel steps in Algorithm 2.2. Let $W_{1,1}$ be the number of operations performed at step 1, and let $W_{i,m}$ be the number of operations performed at step i in Algorithm 2.2 during the mth iteration, $i = 2, 3$. Then, $W_{1,1} = n$, $W_{2,m} = n/2^m = 2^{k-m}$ for $1 \leq m \leq k$, and $W_{3,m} = 2^m$ for $0 \leq m \leq k$. The total number of operations is given by $W(n) = W_{1,1} + \sum_{m=1}^{k} W_{2,m} + \sum_{m=0}^{k} W_{3,m} = n + 2^k \sum_{m=1}^{k} 2^{-m} + \sum_{m=0}^{k} 2^m = n + n(1 - (1/n)) + 2n - 1 = O(n)$. The processor allocation problem is addressed next.

Let our PRAM have $p = 2^q \leq n$ processors $P_1, P_2, \ldots, P_p$, and let $l = n/p = 2^{k-q}$. The input array is divided into p subarrays such that processor P_s is responsible for processing the sth subarray $A(l(s-1)+1), A(l(s-1)+2), \cdots, A(ls)$. At each height h of the binary tree, during either a forward or a backward traversal, the generation of the $B(h, \cdot)$ and $C(h, \cdot)$ values is divided in a similar way among the p processors. This division is performed in a way similar to the details we worked out for the parallel algorithm to compute the sum of n numbers in Example 1.13.

In Algorithm 2.3, which follows, conditions 2.1 and 3.1 hold whenever the number of operations at that particular iteration is greater than or equal

to p. These operations are then distributed equally among the p processors. In the case where the number of operations is less than p (steps 2.2 and 3.2), we assign one operation per processor starting from the lowest-indexed processor. Note that, for any given value of h in the loops defined in steps 2 and 3, the possible number of concurrent operations is $n/2^h = 2^{k-h}$.

The algorithm to be executed by the sth processor is given next. Figure 2.4 illustrates the corresponding processor allocation in the case of $n = 8$ and $p = 2$.

ALGORITHM 2.3

(Algorithm for Processor P_s)

Input: *An array A of size $n = 2^k$, and an index s that satisfies $1 \leq s \leq p = 2^q$, where $p \leq n$ is the number of processors.*

Output: *The prefix sums $C(0, j)$ for $\frac{n}{p}(s-1)+1 \leq j \leq \frac{n}{p}s$.*

begin

1. **for** $j = 1$ **to** $l = n/p$ **do**
   Set $B(0, l(s-1)+j) := A(l(s-1)+j)$
2. **for** $h = 1$ **to** k **do**
   2.1. **if** $(k - h - q \geq 0)$ **then**
   for $j = 2^{k-h-q}(s-1)+1$ **to** $2^{k-h-q}s$ **do**
   Set $B(h, j) := B(h-1, 2j-1) * B(h-1, 2j)$
   2.2. **else** {**if** $(s \leq 2^{k-h})$ **then**
   Set $B(h, s) := B(h-1, 2s-1) * B(h-1, 2s)$}
3. **for** $h = k$ **to** 0 **do**
   3.1. **if** $(k - h - q \geq 0)$ **then**
   for $j = 2^{k-q-h}(s-1)+1$ **to** $2^{k-q-h}s$ **do**

$$\begin{cases} j \text{ even} & : \text{Set } C(h, j) := C\left(h+1, \frac{j}{2}\right) \\ j = 1 & : \text{Set } C(h, 1) := B(h, 1) \\ j \text{ odd} > 1 & : \text{Set } C(h, j) := C\left(h+1, \frac{j-1}{2}\right) * B(h, j) \end{cases}$$

   3.2. **else** $\Big\{$**if** $(s \leq 2^{k-h})$ **then**

$$\begin{cases} s \text{ even} & : \text{Set } C(h, s) := C\left(h+1, \frac{s}{2}\right) \\ s = 1 & : \text{Set } C(h, 1) := B(h, 1) \\ s \text{ odd} > 1 & : \text{Set } C(h, s) := C\left(h+1, \frac{s-1}{2}\right) * B(h, s) \end{cases}\Big\}$$

end

EXAMPLE 2.3:

Let $n = 8$ and let $p = 2$. Consider the algorithm corresponding to processor P_2. At step 1, P_2 sets $B(0, 5) = A(5)$, $B(0, 6) = A(6)$, $B(0, 7) = A(7)$, and $B(0, 8) = A(8)$ (see Figs. 2.2 and 2.4). During the execution of step 2 of

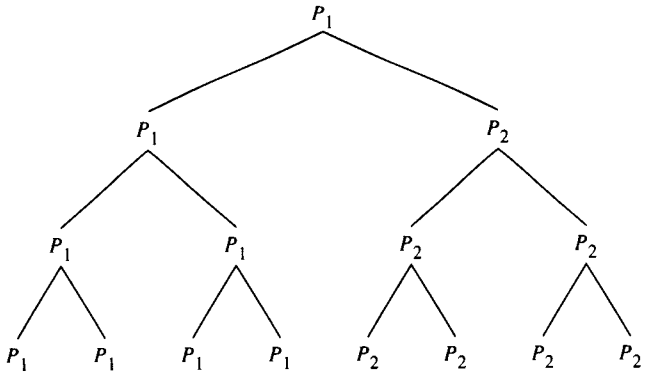


FIGURE 2.4
Processor allocation for computing the prefix sums of eight elements with two processors. The processor indicated at each node is responsible for computing the data generated at that node.

Algorithm 2.3, P_2 will be active for $h = 1, 2$ and idle for $h = 3$. Processor P_2 generates $B(1, 3)$, $B(1, 4)$ and $B(2, 2)$ during the loop defined by step 2. Similarly, during the backward traversal of the binary tree, P_2 will be idle for $h = 3$, and active for $h = 2, 1, 0$. Hence, P_2 generates the sums $C(2, 2)$, $C(1, 3)$, $C(1, 4)$, $C(0, 5)$, $C(0, 6)$, $C(0, 7)$, and $C(0, 8)$ (see Fig. 2.3). □

2.1.4 REVIEW

The basic scheme to build a balanced binary tree on the inputs and to traverse the binary tree to or from the root leads to efficient algorithms for many simple problems. This scheme is one of *the most elementary and the most useful parallel techniques*. We shall use this technique frequently in the rest of this book. Broadcasting a value to all the processors, and compacting the labeled elements of an array, are two simple examples that can be handled efficiently by this scheme (see Exercises 1.8 and 2.3).

The scheme using the balanced binary tree can be generalized to arbitrary balanced trees, where the number of children of an internal node could be nonconstant. As in the binary tree case, a fast algorithm is needed to determine the data stored in an internal node from the data stored in that node's children. It turns out that such a strategy can be carried out successfully for a number of specialized combinatorial problems. Computing the maximum of n elements is one such example discussed in Section 2.6.

2.2 Pointer Jumping

A **rooted-directed tree** T is a directed graph with a special node r such that (1) every $v \in V - \{r\}$ has outdegree 1, and the outdegree of r is 0, and (2) for every $v \in V - \{r\}$, there exists a directed path from v to r. The special node r is called the **root** of T. It follows that a rooted directed tree is a directed graph whose undirected version is a rooted tree such that each arc of T is directed from a node to that node's parent.

The *pointer jumping technique* allows the fast processing of data stored in the form of a set of rooted-directed trees. This technique is best introduced with an example.

2.2.1 FINDING THE ROOTS OF A FOREST

Let F be a forest consisting of a set of rooted directed trees. The forest F is specified by an array P of length n such that $P(i) = j$ if (i, j) is an arc in F; that is, j is the parent of i in a tree of F. For simplicity, if i is a root, we set $P(i) = i$. *The problem is to determine the root $S(j)$ of the tree containing the node j, for each j between 1 and n.*

A simple sequential algorithm—say, one based on first identifying the roots and reversing the links of the trees, followed by performing a depth-first or breadth-first traversal of each tree from its root—solves this problem in linear time. Our fast parallel algorithm follows a completely different strategy.

Initially, the *successor* $S(i)$ of each node i is defined to be the node $P(i)$. The technique of **pointer jumping** (or **path doubling**) consists of *updating the successor of each node by that successor's successor.* As the technique is applied repeatedly, the successor of a node is an ancestor that becomes closer and closer to the root of the tree containing that node. As a matter of fact, the distance between a node and its successor *doubles* unless the successor of the successor node is a root. Hence, after k iterations, the distance between i and $S(i)$ as they appear in a directed tree of F is 2^k unless $S(i)$ is a root. The detailed algorithm is given next.

ALGORITHM 2.4
(Pointer Jumping)
Input: *A forest of rooted directed trees, each with a self-loop at its root, such that each arc is specified by $(i, P(i))$, where $1 \leq i \leq n$.*
Output: *For each vertex i, the root $S(i)$ of the tree containing i.*
begin
1. **for** $1 \leq i \leq n$ **pardo**

Set $S(i)$: $= P(i)$
while $(S(i)) \neq S(S(i))$ **do**
Set $S(i)$: $= S(S(i))$
end

EXAMPLE 2.4:

An illustration of the pointer jumping algorithm is shown in Fig. 2.5. In this case, the forest consists of two trees: one is rooted at vertex 8, and the other is rooted at vertex 13. The arcs in the figure correspond to $(i, P(i))$, $1 \leq i \leq 13$. The first execution of the **while** loop causes vertices 1, 2, 3, 4, 5, 9, 10, and 11 to change their successors, as indicated in Fig. 2.5(b). The second iteration causes the two trees to become of depth 1. We now have $S(i) = S(S(i))$, for all i; hence, the algorithm terminates. □

Let h be the maximum height of any tree in the forest. We can show the correctness of the previous procedure by using an inductive proof on h.

As for the time analysis, the distance (number of arcs in the initial trees) between i and $S(i)$ doubles after each iteration until $S(S(i))$ is the root of the tree containing i. Hence, the number of iterations is $O(\log h)$. Each iteration can be executed in $O(1)$ parallel time with $O(n)$ operations. Therefore, the algorithm runs in $O(\log h)$ time and uses a total of $W(n) = O(n \log h)$ operations, which is clearly nonoptimal (unless h is a constant), since a linear-time sequential algorithm exists.

Theorem 2.2: *Given a forest of rooted directed trees, Algorithm 2.4 generates for each vertex i the root of i's tree. This algorithm runs in* $O(\log h)$ *time using a total number of* $O(n \log h)$ *operations, where h is the maximum height of any tree in the forest, and n is the total number of vertices in the forest.* □

PRAM Model: Algorithm 2.4 requires concurrent-read capability because different nodes could have the same S value. The first iteration of Example 2.4 requires, for instance, that we set $S(3) = S(6)$, $S(4) = S(6)$, and $S(5) = S(6)$; hence, the datum $S(6)$ is used for the execution of three operations. However, no concurrent-write capability is needed. Hence, Algorithm 2.4 is a CREW PRAM algorithm. □

2.2.2 PARALLEL PREFIX

Let us assume next that each node i in the forest F contains a weight $W(i)$. The pointer jumping technique can be also used to compute, for each node i, the sum of the weights stored in the nodes on the path from the node i to the root

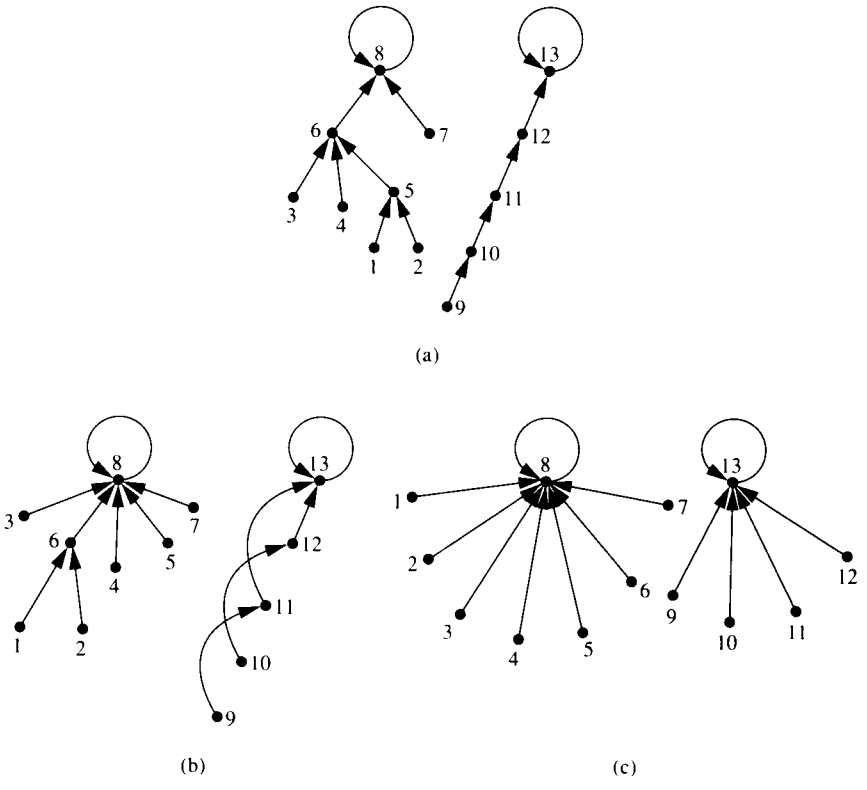


FIGURE 2.5
Illustration of pointer jumping. (a) The initial forest; (b) the forest after the first iteration; (c) the forest after the second iteration.

of i's tree. This computation amounts to generating the prefix sums of several sequences of elements such that each sequence is given by the order of the nodes as they appear on each path.

The next algorithm provides the details.

ALGORITHM 2.5

(Parallel Prefix on Rooted Directed Trees)

Input: *A forest of rooted directed trees, each with a self-loop at its root such that (1) each arc is specified by* $(i, P(i))$, *(2) each vertex* i *has a weight* $W(i)$, *and (3) for each root* r, $W(r) = 0$.

Output: *For each vertex i, $W(i)$ is set equal to the sum of the weights of vertices on the path from i to the root of its tree.*

begin

1. **for** $1 \leq i \leq n$ **pardo**

Set $S(i)$: $= P(i)$

while $(S(i)) \neq S(S(i))$ **do**

Set $W(i)$: $= W(i) + W(S(i))$

Set $S(i)$: $= S(S(i))$

end

EXAMPLE 2.5:

Let us go back to the example of Fig. 2.5. Suppose that, initially, $W(i) = 1$ for all $i \neq 8, 13$, and $W(8) = W(13) = 0$. During the first iteration of the loop, we obtain $W(1) = W(2) = W(3) = W(4) = W(5) = W(9) = W(10) = W(11) = 2$. The other vertices retain their initial W values. Now consider what happens to vertices 1 and 9 during the second iteration. Their associated values change as follows: $W(1) = W(1) + W(6) = 3$, and $W(9) = W(9) + W(11) = 4$. Similarly, we obtain that $W(2) = W(10) = 3$, and $W(3) = W(4) = W(5) = W(11) = 2$, and $W(6) = W(7) = W(12) = 1$. Hence, each $W(i)$ value corresponds to the length of the path from i to the root of its tree; that is, $W(i)$ is equal to the level of i, assuming the root is at level 0. □

PRAM Model: As in Algorithm 2.4, Algorithm 2.5 runs on the CREW PRAM. The bounds on the resources are also asymptotically the same as those of Algorithm 2.4. □

Remark 2.1: An important special case is when each tree is just a path represented by a **linked list.** The problem of computing prefix sums on linked lists is called **parallel prefix,** and is considered in detail in Chapter 3. We note here that Algorithm 2.5 provides a solution to the parallel prefix problem with n nodes such that the running time is $O(\log n)$ and the total number of operations is $O(n \log n)$, since the maximum height h is equal to n in this case. □

2.2.3 REVIEW

The pointer jumping technique provides a simple and powerful method for processing data stored in linked lists or directed rooted trees. In spite of the fact that Algorithms 2.4 and 2.5 are nonoptimal, the pointer jumping technique is useful in general because it is simple and can effectively handle subproblems arising in many computational tasks. These subproblems are usually of a size small enough that the pointer jumping technique will allow optimal overall processing. It is also possible to use the pointer jumping

technique in combination with other techniques to achieve optimality. In the next chapter, we present such an algorithm that solves the parallel prefix problem on linked lists optimally. The pointer jumping technique will be used heavily in Chapter 5.

2.3 Divide and Conquer

The basic **divide-and-conquer** strategy consists of three main steps. *The first step is to partition the input into several partitions of almost equal sizes. The second step is to solve recursively the subproblem defined by each partition of the input.* Note that these subproblems can be solved concurrently in the parallel framework. *The third step is to combine or merge the solutions of the different subproblems into a solution for the overall problem.*

The success of such a strategy depends on whether or not we can perform the first and third steps efficiently. This strategy has been shown to be effective in the development of fast sequential algorithms. It also leads to the most natural way of exploiting parallelism. We illustrate this technique on the convex-hull problem.

2.3.1 THE CONVEX-HULL PROBLEM

Given a set $S = \{p_1, p_2, \ldots, p_n\}$ of n points in the plane, each represented by its (x, y) coordinates, the **planar convex hull** of S is the smallest convex polygon containing all the n points of S. A polygon Q is **convex** if, given any two points p and q in Q, the line segment whose endpoints are p and q lies entirely in Q. The convex-hull problem is to determine the ordered (say, clockwise) list $CH(S)$ of the points of S defining the boundary of the convex hull of S.

EXAMPLE 2.6:

Consider the set S of points shown in Fig. 2.6. In this case, the convex hull of S is given by $CH(S) = (v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8)$. □

The convex-hull problem is an important basic problem in computational geometry that arises in a variety of contexts. Chapter 6 is devoted to the study of several basic problems in planar computational geometry.

A simple divide-and-conquer strategy can be used to solve the convex-hull problem. This approach leads to an $O(n \log n)$ time sequential algorithm. On the other hand, it is not difficult to show that sorting can be reduced to the

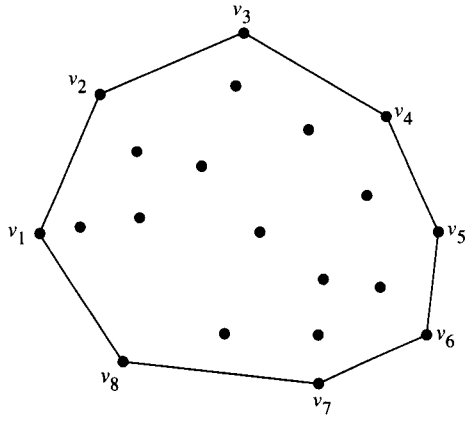


FIGURE 2.6
The convex hull of the set of points shown is the ordered list (v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8).

problem of solving the convex-hull problem (see Exercise 2.16); therefore the sequential complexity of the convex-hull problem is $T^*(n) = \Theta(n \log n)$.

Our parallel algorithm, based on the same divide-and-conquer strategy, will use an optimal number of operations.

2.3.2 A PARALLEL ALGORITHM FOR THE CONVEX-HULL PROBLEM

Let p and q be the points of S with the smallest and the largest x coordinates, respectively. Clearly, p and q belong to $CH(S)$ and partition $CH(S)$ into an **upper hull** $UH(S)$ consisting of all points from p to q of $CH(S)$ (clockwise), and a **lower hull** $LH(S)$ defined similarly from q to p. The upper and the lower hulls of the set of points introduced in Example 2.6 are given by $UH(S) = (v_1, v_2, v_3, v_4, v_5)$ and $LH(S) = (v_5, v_6, v_7, v_8, v_1)$.

For the remainder of this section, we concentrate on determining $UH(S)$. The problem of computing $LH(S)$ can be solved in a similar fashion. Before proceeding, we need the fact stated in the next remark.

Remark 2.2: *Sorting n numbers can be done in $O(\log n)$ time on the EREW PRAM using a total of $O(n \log n)$ operations.* This important fact is established in Chapter 4 for the CREW PRAM model. In a few instances, we use this fact before Chapter 4. □

Assume for simplicity that no two points have the same x or y coordinates and that n is a power of 2.

We start by sorting the points p_i by their x coordinates. By Remark 2.2, this preprocessing step can be performed in $O(\log n)$ time using $O(n \log n)$ operations. Let $x(p_1) < x(p_2) < \cdots < x(p_n)$, where $x(p_i)$ is the x coordinate of p_i.

Let $S_1 = (p_1, p_2, \ldots, p_{\frac{n}{2}})$ and $S_2 = (p_{\frac{n}{2}+1}, \ldots, p_n)$. Suppose that $UH(S_1)$ and $UH(S_2)$ have been determined. The **upper common tangent** between $UH(S_1)$ and $UH(S_2)$ is the common tangent such that both $UH(S_1)$ and $UH(S_2)$ are below it.

EXAMPLE 2.7:

Consider the two upper hulls $UH(S_1)$ and $UH(S_2)$ in Fig. 2.7. In this case, the segment (a, b) is the upper common tangent. □

Determining the upper common tangent between $UH(S_1)$ and $UH(S_2)$ can be done in $O(\log n)$ sequential time by using a binary search method. We shall not elaborate on this further since, in Chapter 6, we present in detail a faster parallel algorithm. However, *we assume for the remainder of this section that we have an $O(\log n)$-time sequential procedure to compute the upper common tangent.*

Let $UH(S_1) = (q_1, \cdots, q_s)$ and $UH(S_2) = (q'_1, \cdots, q'_t)$ be the upper hulls of S_1 and S_2, respectively, given in a left-to-right order. Note that, in particular, $q_1 = p_1$ and $q'_t = p_n$. Suppose that the upper common tangent has

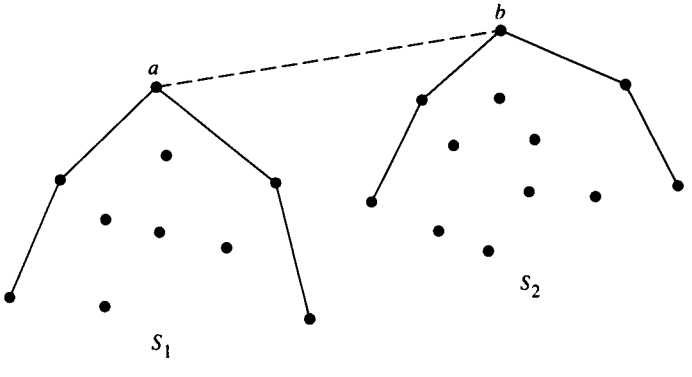


FIGURE 2.7
The line segment (a, b) is the upper common tangent of the upper hulls of S_1 and S_2.

been determined and is given by (q_i, q'_j). Then, $UH(S)$ is the array consisting of the first i entries of $UH(S_1)$ and the last $t - j + 1$ entries of $UH(S_2)$; that is, $UH(S) = (q_1, \ldots, q_i, q'_j, \ldots, q'_t)$. If s and t are given, once the indices i and j are known, $UH(S)$ and its size can be determined in $O(1)$ parallel time using a total of $O(n)$ operations.

PRAM Model: The method just given to combine $UH(S_1)$ and $UH(S_2)$ into $UH(S)$ requires a concurrent-read capability. The requirement is due to the fact that indices i and j defining the upper common tangent are needed to determine whether a point of $UH(S_1)$ or $UH(S_2)$ belongs to $UH(S)$, and, if it does, where it fits.

For example, in the time-processors framework, we can assign a processor to determine an entry of $UH(S)$ as follows. Processor P_k sets $UH(S)(k) := q_k$ whenever $k \leq i$, and sets $UH(S)(k) := q'_{k+j-i-1}$ whenever $i < k \leq i + t - j + 1$. In this case, a processor P_k may have to access i, j, and t. Hence, a CREW PRAM can be used to execute this step within the stated time bound. □

The overall algorithm is given next.

ALGORITHM 2.6
(Simple Upper Hull)
Input: *A set S of n points in the plane, no two of which have the same x or y coordinates such that $x(p_1) < x(p_2) < \cdots < x(p_n)$, where n is a power of 2.*
Output: *The upper hull of S.*
begin
1. If $n \leq 4$, then use a brute-force method to determine $UH(S)$, and exit.
2. Let $S_1 = (p_1, p_2, \cdots, p_{\frac{n}{2}})$ and $S_2 = (p_{\frac{n}{2}+1}, \ldots, p_n)$. Recursively, compute $UH(S_1)$ and $UH(S_2)$ in parallel.
3. Find the upper common tangent between $UH(S_1)$ and $UH(S_2)$, and deduce the upper hull of S.
end

Theorem 2.3: *Algorithm 2.6 correctly computes the upper hull of n points in the plane. This algorithm runs in $O(\log^2 n)$ time using a total of $O(n \log n)$ operations.*

Proof: The correctness proof follows from a simple induction on n, assuming the correctness of the procedure to determine the upper common tangent.

Suppose that Algorithm 2.6 takes $T(n)$ parallel steps using $W(n)$ operations on an arbitrary instance of size n. We define each of $T(n)$ and $W(n)$ by a recurrence relation as follows.

Step 1 takes $O(1)$ sequential time. Step 2 takes $T(n/2)$ time using $2W(n/2)$ operations. As for step 3, determining the upper common tangent can be done in $O(\log n)$ sequential time, as indicated previously, and the combination of $UH(S_1)$ and $UH(S_2)$ into $UH(S)$ can be performed in $O(1)$ parallel time using a total of $O(n)$ operations. Therefore, $T(n)$ and $W(n)$ satisfy the following recurrences:

$$T(n) \leq T\left(\frac{n}{2}\right) + a \log n$$

$$W(n) \leq 2W\left(\frac{n}{2}\right) + bn,$$

where a and b are positive constants. The solutions of these recurrence equations are given by $T(n) = O(\log^2 n)$ and $W(n) = O(n \log n)$, and the theorem follows. □

Corollary 2.1: *The convex hull of a set of n points can be determined in $O(\log^2 n)$ time using a total of $O(n \log n)$ operations. Hence, Algorithm 2.6 is optimal.*

PRAM Model: Algorithm 2.6 requires the CREW PRAM model, since our implementation of the merging of $UH(S_1)$ and $UH(S_2)$ does. In Exercise 2.17, the reader is asked to adapt this algorithm to run on the EREW PRAM model.□

Remark 2.3: In the time-processors framework, the running time of Algorithm 2.6 is $O\left(\frac{n \log n}{p} + \log^2 n\right)$, where p is the number of processors. Therefore, this algorithm achieves an optimal speedup for all values of p such that $p \leq n/\log n$. □

2.3.3 REVIEW

The divide-and-conquer strategy constitutes a powerful, widely applicable approach for developing efficient parallel algorithms. However, a straightforward divide-and-conquer approach does not lead to optimal $O(\log n)$ time algorithms, unless the merging can be performed efficiently. As a matter of fact, we shall see in Chapter 6 how to find the upper common tangent in $O(1)$ parallel time using only a total of $O(n)$ operations. This improvement implies an optimal $O(\log n)$ time algorithm for the convex-hull problem. If the merging step of a divide-and-conquer algorithm proves difficult to speed up, the following two methods provide alternatives.

The first method uses an **n^α divide-and-conquer** (for some $0 < \alpha < 1$) **strategy** by partitioning the input into n^α approximately equal partitions, rather than into a small number of partitions (as we did in the previous example). This strategy can indeed be used to obtain a solution to the convex-hull problem in $O(\log n)$ time using a total of $O(n \log n)$ operations (see Exercise 2.18).

The second approach is to try to **pipeline** the operations required for the merging steps. If successful, such a strategy may enable us to perform merging in $O(1)$ time using a linear number of operations. This **pipelined** or **cascading divide-and-conquer strategy** is used to derive the optimal $O(\log n)$ time merge-sort algorithm (Chapter 4), and to obtain the WT optimal algorithms for several computational geometry problems (Chapter 6). A simple use of the pipelining technique is discussed in Section 2.5.

2.4 Partitioning

The **partitioning strategy** *consists of (1) breaking up the given problem into p independent subproblems of almost equal sizes, and then (2) solving the p subproblems concurrently*, where p is the number of processors available. In its simplest form, this strategy amounts to splitting the input into p nonoverlapping pieces, and then solving the problems associated with the p pieces concurrently. However, in most cases, we make an effort to break up the given problem into a set of independent subproblems as illustrated by the problem of merging two sorted sequences, which we address next.

Given a set S with a partial order relation $\leq$ (that is, $\leq$ is reflexive, antisymmetric, and transitive), S is **linearly ordered** or **totally ordered,** if for every pair $a, b \in S$, either $a \leq b$ or $b \leq a$.

Let $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_n)$ be two nondecreasing sequences whose elements are drawn from a linearly ordered set S. We consider, in this section, the problem of **merging** these two sequences into a sorted sequence—say, $C = (c_1, c_2, \ldots, c_{2n})$.

Simple linear-time sequential algorithms to solve the merging problem are well known. Our goal here is to provide a parallel solution that is based on *partitioning A and B into many pairs of subsequences such that we can obtain the sorted sequence by concurrently merging the resulting pairs of subsequences.*

Compare this strategy with the divide-and-conquer strategy, where the main work required typically lies in combining the solutions of the subproblems involved, rather than in partitioning the input.

2.4.1 A SIMPLE MERGING ALGORITHM

We start with few definitions. Let $X = (x_1, x_2, \ldots, x_t)$ be a sequence whose elements are from the set S. Let $x \in S$. The **rank** of x in X, denoted by $rank(x : X)$, is the number of elements of X that are less than or equal to x. Let $Y = (y_1,$

$y_2, \ldots, y_s)$ be an arbitrary array of elements from S. **Ranking** Y in X is the problem of determining the array $rank(Y:X) = (r_1, r_2, \ldots, r_s)$, where $r_i = rank(y_i : X)$.

EXAMPLE 2.8:

Let $X = (25, -13, 26, 31, 54, 7)$ and $Y = (13, 27, -27)$. Then, $rank(Y:X) = (2, 4, 0)$. □

Without loss of generality, assume that all the elements appearing in the two sorted sequences to be merged, A and B, are *distinct* (see Exercise 2.21). In particular, no element of A appears in B.

The merging problem can be viewed as that of determining the rank of each element x from A or B in the set $A \cup B$. If $rank(x : A \cup B) = i$, then $c_i = x$, where c_i is the ith element of the desired sorted sequence. Since $rank(x:A \cup B) = rank(x:A) + rank(x:B)$, we can solve the merging problem by determining the two integer arrays $rank(A : B)$ and $rank(B : A)$.

We now proceed to describe an algorithm to determine $rank(B:A)$. The same algorithm can be used to compute $rank(A : B)$. Let b_i be an arbitrary element of B. Since A is sorted, we can find the rank of b_i in A by using the **binary search method.** This method consists of comparing b_i with the middle element of A. Based on the outcome of this comparison, the search can be restricted to the upper or lower half of A. The process is repeated until b_i is isolated between two successive entries of A—that is, $a_{j(i)} < b_i < a_{j(i)+1}$, where $rank(b_i : A) = j(i)$. Note that we have used here the fact that the elements of A and B are distinct.

The binary search algorithm just sketched determines the rank of an arbitrary element of B in A, and runs in $O(\log n)$ sequential time. We can obviously apply this method concurrently to all the elements of B to obtain a parallel algorithm for ranking B in A whose running time is $O(\log n)$, which implies that we have an $O(\log n)$ time parallel algorithm for merging two sequences, each of length n. However, the total number of operations used by such an algorithm is $O(n \log n)$, which is nonoptimal, since linear-time sequential algorithms exist, as indicated at the beginning of this section.

2.4.2 AN OPTIMAL MERGING ALGORITHM

An optimal merging algorithm can be obtained as follows. Choose approximately $n/\log n$ elements of each of A and B that partition A and B into blocks of almost equal lengths. Apply the binary search method to rank each of the chosen elements in the other sequence. This step reduces the problem into

merging pairs of subsequences, each of which has $O(\log n)$ elements. We can then apply an optimal-time sequential algorithm to each pair of subsequences to generate the sorted sequence.

For simplicity, we present the details of a slightly different algorithm. We start with the following partitioning algorithm, which forms the main component of the overall algorithm. In Algorithm 2.7 we do not assume that the lengths of the two sequences are necessarily equal. Figure 2.8 illustrates the pairs of subsequences obtained.

ALGORITHM 2.7

(Partition)

Input: *Two arrays* $A = (a_1, \ldots, a_n)$ *and* $B = (b_1, \ldots, b_m)$ *in increasing order, where both* $\log m$ *and* $k(m) = m/\log m$ *are integers.*
Output: $k(m)$ *pairs* (A_i, B_i) *of subsequences of* A *and* B *such that (1)* $|B_i| = \log m$, *(2)* $\sum_i |A_i| = n$, *and (3) each element of* A_i *and* B_i *is larger than each element of* A_{i-1} *or* B_{i-1}, *for all* $1 \leq i \leq k(m) - 1$.

begin

1. *Set* $j(0) := 0, j(k(m)) := n$
2. **for** $1 \leq i \leq k(m) - 1$ **pardo**
   - 2.1. *Rank* $b_{i \log m}$ *in* A *using the binary search method, and let* $j(i) = rank(b_{i \log m} : A)$
3. **for** $0 \leq i \leq k(m) - 1$ **pardo**
   - 3.1. *Set* $B_i := (b_{i \log m + 1}, \ldots, b_{(i+1) \log m})$
   - 3.2. *Set* $A_i := (a_{j(i)+1}, \ldots, a_{j(i+1)})$

   (A_i *is empty if* $j(i) = j(i + 1)$)

end

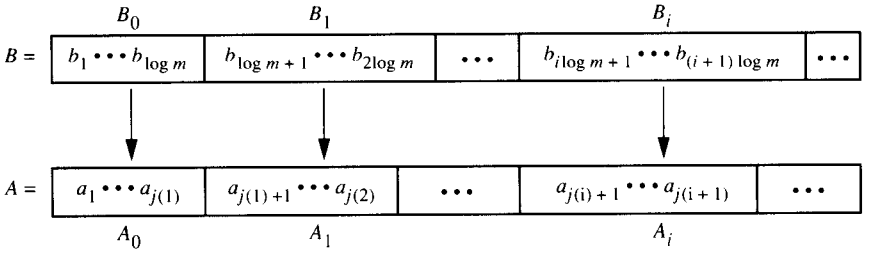


FIGURE 2.8
The partitions generated by the partition algorithm (Algorithm 2.7). Each B_i is of size $\log m$, and the A_i's could be of different sizes. The element $j(i)$ is given by $j(i) = rank(b_{i \log m} : A)$.

EXAMPLE 2.9:

Let $A = (4, 6, 7, 10, 12, 15, 18, 20)$ and let $B = (3, 9, 16, 21)$. In this case, $m = 4$ and $k(m) = 2$. Since $rank(9 : A) = 3$, we obtain the following two pairs of subsequences: (1) $A_0 = (4, 6, 7)$ and $B_0 = (3, 9)$, and (2) $A_1 = (10, 12, 15, 18, 20)$ and $B_1 = (16, 21)$. Each element of A_1 and B_1 is larger than each element in A_0 or B_0; hence, we can merge A and B by merging separately the pairs (A_0, B_0) and (A_1, B_1). □

Lemma 2.1: *Let C be the sorted sequence obtained by merging the two sorted sequences A and B, of lengths n and m, respectively. Then, Algorithm 2.7 partitions A and B into pairs of subsequences (A_i, B_i) such that $|B_i| = O(\log m)$, $\sum_i |A_i| = n$, and $C = (C_0, C_1, \cdots)$, where C_i is the sorted sequence obtained by merging A_i and B_i. Moreover, this algorithm runs in $O(\log n)$ time using a total of $O(n + m)$ operations.*

Proof: We first establish that each element in the subsequences A_i and B_i is larger than each element of A_{i-1} or B_{i-1}, for $1 \leq i \leq k(m) - 1$. The two smallest elements of A_i and B_i are, respectively, $a_{j(i)+1}$ and $b_{i \log m + 1}$, whereas the two largest elements of A_{i-1} and B_{i-1} are, respectively, $a_{j(i)}$ and $b_{i \log m}$. Since $rank(b_{i \log m} : A) = j(i)$, we have that $a_{j(i)} < b_{i \log m} < a_{j(i)+1}$. This result implies that $b_{i \log m + 1} > b_{i \log m} > a_{j(i)}$, and $a_{j(i)+1} > b_{i \log m}$. Therefore, each of the elements of A_i and B_i is larger than each of the elements in A_{i-1} or B_{i-1}; hence, the correctness of the algorithm follows.

The timing analysis can be done as follows. Step 1 takes $O(1)$ sequential time. Step 2 takes $O(\log n)$ time, since the binary search method is applied to all the elements in parallel. The total number of operations required to execute this step is $O((\log n) \times (m/\log m)) = O(m + n)$, since $(m \log n / \log m) < (m \log (n + m)/\log m) \leq n + m$, for $n, m \geq 4$. Step 3 takes $O(1)$ parallel time using a linear number of operations. Hence, the algorithm runs in $O(\log n)$ time using a total of $O(n + m)$ operations. □

After applying Algorithm 2.7 to our merging problem (of two sequences, each of length n), we end up with an independent set of merging subproblems. This outcome is the essence of the partitioning strategy. Now, we would like to handle each merging subproblem in $O(\log n)$ time, such that the total number of operations used is proportional to the size of the subproblem. This running time can be achieved as follows.

Consider the merging subproblem corresponding to an arbitrary pair (A_i, B_i). Recall that $|B_i| = \log n$ for all indices i. If $|A_i| = O(\log n)$, we can merge the pair (A_i, B_i) in $O(\log n)$ sequential time by using an optimal sequential algorithm. Otherwise, we can apply Algorithm 2.7 to partition A_i into blocks each of which is of size $O(\log n)$ (in this case, A_i plays the role of B, and B_i plays the role of A). This step will take $O(\log \log n)$ time using $O(|A_i|)$ operations.

Therefore, we can make each of the subsequences to be of length $O(\log n)$ without asymptotically increasing the bounds on the resources. Finally, an optimal sequential algorithm for merging can be applied to each pair of subsequences to generate the desired sorted sequence. We therefore have the following theorem.

Theorem 2.4: *Let A and B be two sorted sequences, each of length n. Merging A and B can be done in $O(\log n)$ time, using a total of $O(n)$ operations.* □

PRAM Model: The binary search method is applied to the array A to rank several elements of B simultaneously in step 2 of Algorithm 2.7. Hence, concurrent-read capability is required by this algorithm; however, no concurrent-write capability is needed. The algorithm thus runs on the CREW PRAM model. □

2.4.3 REVIEW

We have presented a partitioning strategy to solve the merging problem efficiently. In Chapter 4, we shall see how to use this strategy with a parallel-search algorithm (as opposed to the binary search algorithm used before) to obtain an optimal merging algorithm that runs in $O(\log \log n)$ time. This algorithm is one of the fast optimal algorithms that does not require concurrent-write capability.

The partitioning strategy used in this section should be contrasted with the divide-and-conquer strategy. Both strategies have the same goal of decomposing the problem into a set of subproblems that can be solved in parallel. The main work in the divide-and-conquer strategy usually lies in the merging of the solutions of the subproblems, whereas the main work in the partitioning strategy lies in carefully decomposing the problem so that the solutions of the subproblems can be combined easily to generate the solution of the overall problem.

2.5 Pipelining

A **2-3 tree** is a rooted tree in which each internal node has two or three children, and every path from the root to a leaf is of the same length. The number of leaves in a 2-3 tree of height h is between 2^h and 3^h (see Exercise 2.27). Hence, if the number of leaves is n, the height of the tree is $\Theta(\log n)$.

A sorted list A of n items, $A = (a_1, a_2, \ldots, a_n)$, where $a_1 < a_2 < \cdots < a_n$, can be represented by a 2-3 tree T, where the leaves hold the data items in a left-to-right order. An internal node v will hold two data items, $L[v]$ and $M[v]$, where $L[v]$ and $M[v]$ are, respectively, the largest items stored in the first (leftmost) and the second subtrees of v. In addition, v holds a data item $R[v]$ representing the largest item in the third (rightmost) subtree of v, whenever v has a third child. The item $R[v]$ usually is not included in the definition of a 2-3 tree. We shall make use of it in a parallel insertion procedure introduced later in this section.

EXAMPLE 2.10:

Let A be the list $A = (1, 2, 3, 4, 5, 6, 7)$. A 2-3 tree representing A is shown in Fig. 2.9. We have used the notation $i : j : k$ next to each internal node v to indicate that $L[v] = i$, $M[v] = j$ and $R[v] = k$, whenever a rightmost subtree exists. Note that the 2-3 tree corresponding to A is not unique. □

2.5.1 BASIC OPERATIONS ON A 2-3 TREE

A 2-3 tree is a well-known data structure used to represent sets of elements subject to the operations of search, insert, and delete. More precisely, a 2-3 tree represents a dynamically changing set of elements induced by processing a sequence of instructions containing the operations of (1) searching for an element, (2) inserting an element, and (3) deleting an element, where the elements are drawn from a linearly ordered set. Notice that the set of elements represented by a 2-3 tree appear in sorted order, as discussed before,

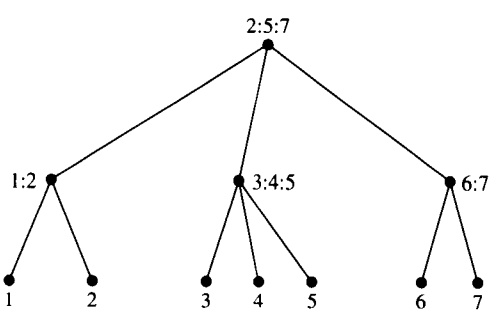


FIGURE 2.9
An example of a 2-3 tree. The notation $L[v] : M[v] : R[v]$ next to node v denotes respectively, the largest elements in the leftmost, the middle, and the rightmost (if it exists) subtrees.

and that the insert and delete operations require that the structure of the 2-3 tree be preserved after the operations are executed. Each of the search, insert, and delete operations can be performed in $O(\log n)$ time, where n is the size of the current set of elements. We next give a brief overview of the procedures related to the search and insert operations.

Let T be a 2-3 tree representing a list of n elements, $a_1 < a_2 < \cdots < a_n$. Given an element b, the problem of searching for b can be defined as the problem of finding the leaf u of T such that the data item a_i stored in u satisfies $a_i \leq b < a_{i+1}$.

This problem can be handled with a binary search method as follows. Let r be the root of T. The search for b can be restricted to one of the subtrees of r, depending on where b fits between the two data items $L[r]$ and $M[r]$. More precisely, if $b \leq L[r]$, the search continues in the leftmost subtree of r; otherwise, if $L[r] < b \leq M[r]$, the search continues in the second subtree. We handle the remaining case of $b > M[r]$ by restricting the search to the third subtree. The search procedure terminates when a leaf is reached. This process takes time proportional to the height of T.

We can insert an element b into the tree T by first applying the previous search procedure to locate b. Let u be the leaf identified by this search procedure. Node u contains an element a_i such that $a_i \leq b < a_{i+1}$. If $b = a_i$, there is nothing to be done, since b is already in the tree. Otherwise, we create a leaf u', which holds the data item b. We insert u' to the immediate right of u with the same parent—say p—as u. If p has three children, we stop here. Thus, let us assume that p has four children. We create a new node p' and allocate the children of p appropriately between p and p' (see Fig. 2.10). We can then insert the node p' into T by using the same procedure. That is, p' is tentatively assigned the same parent as p, which is in turn examined for a

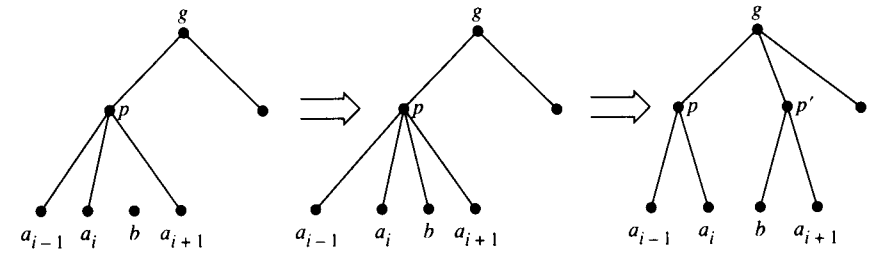


FIGURE 2.10
The process of inserting an item b into a 2-3 tree. A node containing b is created and assigned the same parent p as a_i thereby generating the middle tree. Since node p in the middle tree has four children, a new node p' is created, and is assigned the same parent as p, as illustrated in the rightmost tree.

possible violation of the restriction on the number of children. Finally, if the root r has four children, we create a new root with two children, each of which has the appropriate two children of r.

During the insertion procedure, the L, M, and R values can be adjusted appropriately without asymptotically increasing the sequential time of the procedure.

EXAMPLE 2.11:

Suppose we have to insert the data item 5 into the 2-3 tree T shown in Fig. 2.11(a). A leaf containing 5 is created with the node c as the parent (Fig. 2.11b). Since c has four children, a node c' is created whose children are the leaves containing 5 and 6 (Fig. 2.11c). But now the root has four children. Creating a new root with two children results in the 2-3 tree shown in Fig. 2.11(d). □

2.5.2 INSERTING A SEQUENCE OF ITEMS INTO A 2-3 TREE

Suppose that we are given a 2-3 tree T holding the n items $a_1 < a_2 < \cdots < a_n$. *We consider the problem of inserting a sequence of k items* $b_1 < b_2 < \cdots <$

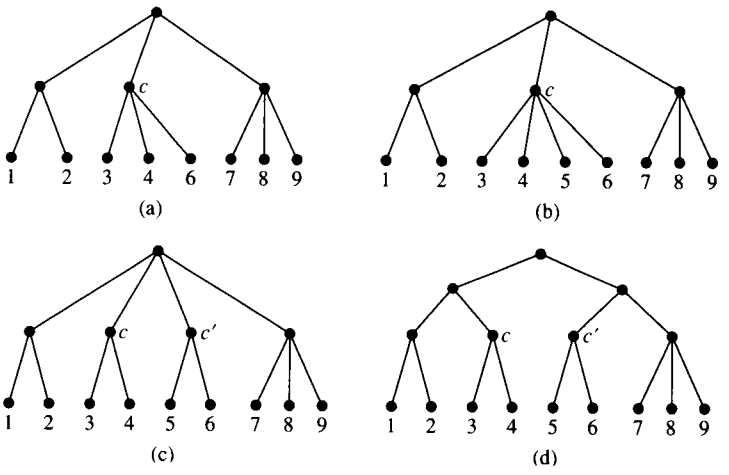


FIGURE 2.11
Steps for inserting data item 5 into a 2-3 tree. (a) A 2-3 tree. (b) and (c) Intermediate trees generated during insertion of the element 5 into the tree. (d) The final tree generated.

b_k *into* T, where k is assumed to be much smaller than n. If k is larger than n, then it is more efficient to construct the corresponding 2-3 tree from scratch (see Exercise 2.32).

Using the previous insertion procedure, we can insert the k elements, one at a time, in $O(k \log n)$ sequential time. In what follows, we describe a parallel algorithm that performs the insertion of the k elements in $O(\log n)$ time using a total of $O(k \log n)$ operations. A key component of this algorithm is the effective *pipelining* of several subsequences of insertions.

Briefly, the technique of **pipelining** *is the process of breaking up a task into a sequence of subtasks—say,* $t_1, \ldots, t_m$*—such that, once* t_1 *is completed, the sequence corresponding to a new task can begin and can proceed at the same rate as the previous task.* This process is similar to the operation of an assembly line, where a subtask could be the completion of a specific component of a system, and several systems are assembled on line.

We start with a preprocessing step that simplifies the presentation of the algorithm. We insert b_1 and b_k into T using the sequential insertion algorithm. This step ensures that each of the remaining b_i's fits between two consecutive leaves of T. This preprocessing takes $O(\log n)$ sequential time. Hence, we assume for the remainder of this section that T has been preprocessed in this fashion.

Concurrently, we locate all the elements b_i in T by applying the search procedure on each b_i separately. Hence, for each b_i we obtain a leaf containing the item $a_{i'}$ such that $a_{i'} \leq b_i < a_{i'+1}$. This search takes $O(\log n)$ parallel steps, using a total of $O(k \log n)$ operations.

PRAM Model: Locating all the b_i's in T requires a concurrent-read capability of the tree T; hence, this procedure runs on the CREW PRAM model. In Exercise 2.28, the reader is asked to show how to achieve the same performance on the EREW PRAM model. □

Let us call a **chain** the ordered set of elements among the b_i's that have to fit between two consecutive leaves of T. In particular, let B_i be the chain that has to fit between a_i and a_{i+1}, and let $|B_i| = k_i$, where $1 \leq i \leq n - 1$. Clearly, $\sum_i k_i = k - 2$ (since b_1 and b_k have been already inserted in T). We first consider the special case when $k_i \leq 1$, for all $1 \leq i \leq n - 1$.

The procedure of inserting the k elements consists of a bottom-up processing of T similar to the sequential insertion procedure, except that several nodes at the same level may be processed concurrently.

Initially, we create a leaf l_i holding b_i, and assign the appropriate parent to it, for each $1 \leq i \leq n - 1$. Some of the internal nodes at height 1 may now have more than three children, but none will have more than six. For each node v of more than three children, we create another node v' with the same parent as v, and reassign the children of v appropriately between v and v'. This step can be done concurrently for all the internal nodes at height 1 with

more than three children. This process can be repeated until we reach the root. The root is then handled just as in the sequential case.

Hence, inserting k elements into T can be performed in $O(\log n)$ time using $O(k \log n)$ operations in the case when $k_i \leq 1$, for $1 \leq i \leq n - 1$. At this point, the reader should verify that the data items stored in the nodes can be updated appropriately as we proceed with the insertion procedure. We call the algorithm just described the **simple parallel insertion algorithm.**

Consider the general case where the size k_i of each B_i is not necessarily less than or equal to 1. We apply the simple parallel insertion algorithm to the middle elements of the nonempty B_i's. The middle element of a nonempty chain $a_l, a_{l+1}, \cdots, a_f$ is the element a_s, where $s = \lceil (f + l)/2 \rceil$. Inserting these elements takes $O(\log n)$ time and, in addition, reduces the size of each chain by a factor of ½. We repeat this process on the newly formed chains. Note that the new chains can be trivially deduced from the old chains. More precisely, the chain $a_l, a_{l+1}, \ldots, a_f$ causes the creation of the two new chains $a_l, \ldots, a_{s-1}$, and $a_{s+1}, \ldots, a_f$, whenever $s - 1 \geq l$ and $s + 1 \leq f$, respectively. After $O(\log k)$ iterations, all the k elements are inserted in T. The overall time is $O(\log k \log n)$, since each iteration takes $O(\log n)$ time. However, we can obtain a faster algorithm by using the technique of *pipelining*.

Let us call each application of the simple parallel insertion procedure a **stage.** Hence, the algorithm consists of $\leq \lceil \log k \rceil$ stages such that, during the ith stage, a set of elements is inserted into the 2-3 tree T_{i-1} to obtain the 2-3 tree T_i, where initially $T_0 = T$. Each T_{i-1} is processed bottom-up, level by level. Hence, stage i can be viewed as a **wave** moving up T_{i-1}. Once the nodes at a certain level are processed, the next wave can move up to process the appropriate nodes at this level. Therefore, the different stages can be pipelined one after the other, up the tree. The running time of this method is reduced to $O(\log n + \log k) = O(\log n)$, using a total of $O(k \log n)$ operations.

PRAM Model: The parallel insertion procedure does not in itself require any simultaneous access to the same memory location. However, the method used to locate the elements b_i in T requires the CREW PRAM model. Hence, the overall insertion procedure runs on the CREW PRAM. □

2.5.3 REVIEW

Pipelining is an important parallel technique that has been used extensively in parallel processing. We shall encounter this technique in the development of an optimal $O(\log n)$ merge-sort algorithm, presented in Chapter 4, and in speeding up the divide-and-conquer algorithms for several problems in computational geometry, presented in Chapter 5.

2.6 Accelerated Cascading

Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of elements drawn from a linearly ordered set S. The problem of computing the **maximum** element of X is to find an element x_i such that $x_i \geq x_j$, for all $1 \leq j \leq n$.

This basic problem can be solved by a simple linear-time sequential algorithm. On the PRAM, it offers an interesting case study for introducing the strategy of *accelerated cascading*.

In the remainder of this section, we assume that the x_i's are distinct. Otherwise, we can replace x_i by the pair (x_i, i), for each i, and extend the relation $\geq$ as follows: $(x_i, i) > (x_j, j)$ if and only if $x_i > x_j$, or $x_i = x_j$ and $i > j$.

We can determine the maximum element by using a balanced binary tree constructed on the n input elements, just as in the case of computing the sum of n elements (Algorithm 1.7). The running time of the parallel algorithm is $O(\log n)$, and the total number of operations used is $O(n)$. This parallel algorithm is optimal, since the work performed matches the sequential complexity of the problem.

We can obtain a *faster* algorithm by using a scheme based on a *doubly logarithmic-depth tree computation* (as opposed to the balanced binary tree of logarithmic depth). The main idea is to construct a balanced tree with the x_i's as the leaves such that the number of children of a node u is equal to $\lceil \sqrt{n_u} \rceil$, where n_u is the number of leaves in the subtree rooted at u. Each internal node will be used to hold the maximum element in its subtree. The condition on the number of children forces the tree to be of doubly logarithmic depth. The success of this strategy depends on the existence of a *constant-time algorithm* to perform the operation represented by an internal node—that is, a constant-time algorithm for computing the maximum of an arbitrary number of elements. Our next task is the development of such an algorithm.

2.6.1 A CONSTANT-TIME ALGORITHM FOR COMPUTING THE MAXIMUM

Let A be an array holding p elements from our linearly ordered universe S. The purpose of Algorithm 2.8 is to perform comparisons between all pairs of elements from A. The maximum element can be identified as the only element that comes out a "winner" in all its comparisons.

In the algorithm that follows, the Boolean array B holds the outcomes of all the comparisons, and the symbol $\wedge$ represents the Boolean AND operation.

ALGORITHM 2.8
(Basic Maximum)
Input: *An array A of p distinct elements.*
Output: *A Boolean array M such that $M(i) = 1$ if and only if $A(i)$ is the maximum element of A.*
begin
1. **for** $1 \leq i, j \leq p$ **pardo if** $(A(i) \geq A(j))$ **then** *set* $B(i, j) := 1$
   else *set* $B(i, j) := 0$
2. **for** $1 \leq i \leq p$ **pardo**
   Set $M(i) := B(i, 1) \wedge B(i, 2) \wedge \cdots \wedge B(i, p)$

end

When the algorithm terminates, $M(i) = 1$ if and only if $A(i)$ is the maximum element.

PRAM Model: Step 1 requires simultaneous read operations. Step 2 can be implemented in $O(1)$ time if we allow concurrent writes of the same value (see Exercise 1.5). Hence, Algorithm 2.8 can be implemented on the common CRCW PRAM in $O(1)$ time using $O(p^2)$ operations. □

Lemma 2.2: *The maximum of p elements can be computed on the common CRCW PRAM in $O(1)$ time using a total of $O(p^2)$ operations.* □

2.6.2 A DOUBLY LOGARITHMIC-TIME ALGORITHM

Given a rooted tree, the **level** of a node u is the number of edges on the path between u and the root of the tree. Hence, the level of the root is 0. We are ready to introduce the doubly logarithmic-depth tree.

For clarity, assume that $n = 2^{2^k}$, for some integer k. Define the **doubly logarithmic-depth tree** with n leaves as follows. The root of the tree has $2^{2^{k-1}}$ (that is, $\sqrt{n}$) children, each of its children has $2^{2^{k-2}}$ children, and, in general, each node at the ith level has $2^{2^{k-i-1}}$ children, for $0 \leq i \leq k-1$. Each node at level k will have two leaves as children.

EXAMPLE 2.12:

The doubly logarithmic-depth tree for the case when $n = 16$ is shown in Fig. 2.12. The root has four children, and each of the other internal nodes has two children. Each internal node corresponds to computing the maximum of that node's children. □

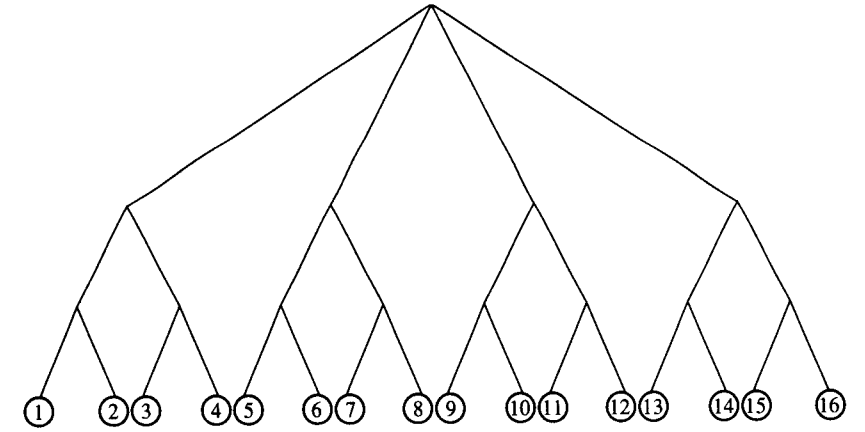


FIGURE 2.12
Doubly logarithmic-depth tree on 16 nodes.

A simple inductive argument on i shows that the number of nodes at the ith level of the doubly logarithmic-depth tree is $2^{2^k - 2^{k-i}}$, for $0 \leq i < k$. The number of nodes at the kth level is $2^{2^{k-1}} = n/2$. In addition, the depth of this tree is $k + 1 = \log\log n + 1$.

The doubly logarithmic-depth tree can be used for computing the maximum of n elements. Each internal node holds the maximum of the elements stored in its subtree. The algorithm proceeds level by level, bottom up, starting with the nodes at height 1. Using Algorithm 2.8, the maxima required at any given level can be computed in $O(1)$ time.

It follows that the maximum can be computed in $T(n) = O(\log\log n)$ time, so the doubly logarithmic-depth tree algorithm is *exponentially* faster than the previous $O(\log n)$ time algorithm (Algorithm 1.7).

We still need to estimate the total number of operations required by the new method. The number of operations required by the tasks at the ith level is $O((2^{2^{k-i-1}})^2)$ per node, for $0 \leq i \leq k$, giving a total of $O((2^{2^{k-i-1}})^2 \cdot 2^{2^k - 2^{k-i}}) = O(2^{2^k}) = O(n)$ operations per level. Hence, the total number of operations required by the overall computation is $W(n) = O(n \log\log n)$, which makes the algorithm nonoptimal.

2.6.3 MAKING THE FAST ALGORITHM OPTIMAL

In summary, we have introduced two algorithms for computing the maximum. The first, based on the logarithmic-depth binary tree, is optimal and runs in

logarithmic time, whereas the second algorithm is nonoptimal but runs in doubly logarithmic time ("very fast"). In such a case, we can try to use a strategy, called **accelerated cascading,** to combine the two algorithms into an optimal and a very fast algorithm. We first describe the accelerated-cascading strategy in general terms:

1. Start with the optimal algorithm until the size of the problem is reduced to a certain threshold value.
2. Then, shift to the fast but nonoptimal algorithm.

We now examine the adaptation of this strategy to the problem of computing the maximum of n elements.

In phase 1, we apply the binary tree algorithm, starting from the leaves and moving up to $\lceil \log \log \log n \rceil$ levels. Since the number of candidates reduces by a factor of ½ per level as we go up the binary tree, we know that the maximum is among the $n' = O(n/\log \log n)$ elements generated at the end of the binary tree algorithm. The total number of operations used so far is $O(n)$, and the corresponding time is $O(\log \log \log n)$.

During phase 2, we use the doubly logarithmic-depth tree based on the n' generated elements during phase 1. This second phase requires $O(\log \log n') = O(\log \log n)$ time and uses $W(n) = O(n' \log \log n') = O(n)$ operations. Therefore, the overall algorithm is optimal and runs in time $O(\log \log n)$.

Theorem 2.5: *Finding the maximum of n elements can be done optimally in* $O(\log \log n)$ *time on the common CRCW PRAM.* □

PRAM Model: The basic maximum algorithm (Algorithm 2.8) requires a concurrent write of the same value capability to run in constant time. Since our analysis is based on this assumption, our optimal $O(\log \log n)$ time algorithm requires the common CRCW PRAM model. A matching lower bound for the CRCW PRAM (regardless of whether it is common, arbitrary, or priority) is derived in Chapter 4 in the case where $O(n)$ processors are available. Therefore, *our* $O(\log \log n)$ *time algorithm is WT optimal for the CRCW PRAM model.*

On the other hand, $\Omega(\log n)$ time is required on the CREW PRAM regardless of the number of processors available (Chapter 10). Hence, *the balanced binary tree method yields a WT optimal EREW and CREW PRAM algorithm for computing the maximum.* □

Remark 2.4: An *important special case* of the accelerated-cascading technique is when the optimal algorithm used in phase 1 is an **optimal sequential** algorithm. In fact, we can obtain an optimal $O(\log \log n)$ time algorithm for computing the maximum of n elements by using this special case.

We partition the input into $n/\log\log n$ blocks $\{B_i\}$, each block containing approximately $\log\log n$ elements. We then use the optimal sequential algorithm to compute the maximum of each block, for all blocks concurrently. We then proceed with the doubly logarithmic-depth tree to generate the maximum within the claimed bounds. □

2.6.4 REVIEW

In this section, two general techniques were used to obtain the fast optimal algorithm to compute the maximum. The first consists of organizing the computation in a doubly logarithmic-depth tree. The second is the accelerated cascading of two algorithms, one optimal but relatively slow, the other very fast but nonoptimal. These two techniques seem to be useful for deriving very fast parallel optimal algorithms.

In an important special case, we (1) partition the input into nonoverlapping pieces, (2) solve the subproblems associated with the pieces concurrently by using an optimal sequential algorithm, and (3) combine the solutions of the subproblems by using a fast parallel algorithm.



2.7 Symmetry Breaking

Let $G = (V, E)$ be a directed cycle; that is, the indegree and the outdegree of each vertex is 1, and, for any two vertices $u, v \in V$, there exists a directed path from u to v. A ***k*-coloring** of G is a mapping $c : V \to \{0, 1, \cdots, k-1\}$, such that $c(i) \neq c(j)$ if $\langle i, j\rangle \in E$. We are interested in the problem of determining a 3-coloring of G.

A straightforward algorithm consists of traversing the cycle from an arbitrary vertex, while assigning alternate colors from $\{0, 1\}$ to adjacent vertices. A third color may be needed to terminate the cycle. This simple sequential algorithm is clearly optimal, but it does not seem to lead to a fast parallel algorithm.

The main difficulty in solving our coloring problem fast using parallelism lies in the apparent symmetry in the problem. Assigning a color to many vertices in parallel implies that these vertices have been somehow distinguished from the remaining vertices. But all vertices look alike; hence, some mechanism should be introduced to partition the vertices into classes such that each class can be assigned the same color.

This symmetric situation occurs in many other problems as well. In the remainder of this section, we present an almost constant-time algorithm to break this symmetry. Another technique for breaking symmetry using *randomization* is presented in Chapter 9.

2.7.1 A BASIC COLORING ALGORITHM

We make the following assumption regarding the input representation of the directed cycle $G = (V, E)$: The arcs of G are specified by an array S of length n such that $S(i) = j$ whenever $\langle i,j \rangle \in E$, for $1 \leq i,j \leq n$. Note that we can obtain the **predecessor** relation immediately by setting $P(S(i)) = i$, for $1 \leq i \leq n$.

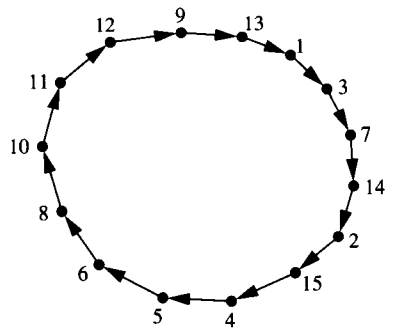
Suppose that an initial coloring c of the vertices of G is given (we can always start with $c(i) = i$ for all i). Given the binary expansion of an integer i—say, $i = i_{t-1} \cdots i_k \cdots i_1 i_0$—the **$k$th least significant bit** of i is bit i_k. We can reduce the number of colors by using the following simple procedure that takes advantage of the binary representation of the colors.

ALGORITHM 2.9
(Basic Coloring)
Input: *A directed cycle whose arcs are specified by an array S of size n and a coloring c of the vertices.*
Output: *Another coloring c' of the vertices of the cycle.*
begin
for $1 \leq i \leq n$ **pardo**
1. Set k to the least significant bit position in which c(i) and c(S(i)) disagree.
2. Set $c'(i) := 2k + c(i)_k$
end

EXAMPLE 2.13:

Let $c(i) = i$ be the initial coloring assigned to the vertices of the directed cycle shown in Fig. 2.13. For each vertex, we list the value of k and its corresponding new color in two separate columns. The number of colors is reduced from 15 to six in this case. □

Lemma 2.3: *The output function c' generated by Algorithm 2.9 is a valid coloring whenever c is a valid coloring. The algorithm runs in* $T(n) = O(1)$ *time using* $W(n) = O(n)$ *operations.*



v	c	k	c'
1	0001	1	2
3	0011	2	4
7	0111	0	1
14	1110	2	5
2	0010	0	0
15	1111	0	1
4	0100	0	0
5	0101	0	1
6	0110	1	3
8	1000	1	2
10	1010	0	0
11	1011	0	1
12	1100	0	0
9	1001	2	4
13	1101	2	5

FIGURE 2.13
The basic coloring algorithm on a directed cycle, where each arc is represented by $\langle i, S(i)\rangle$. The column c represents the initial coloring $c(i) = i$, k is the least significant bit position in which $c(i)$ and $c(S(i))$ differ, and $c'(i) = 2k + c(i)_k$ is the new coloring. The vertices $\{v\}$ are listed in the order they appear on the cycle.

Proof: Notice that, in the basic coloring procedure, the index k must always exist, since c is a coloring. Suppose that $c'(i) = c'(j)$ for some $\langle i,j\rangle \in E$. Then, $c'(i) = 2k + c(i)_k$ and $c'(j) = 2l + c(j)_l$, where k and l are determined as stated in the basic coloring algorithm. Since $c'(i) = c'(j)$, we must have $k = l$. But then $c(i)_k = c(j)_k$, which contradicts the definition of k. Hence, $c'(i) \neq c'(j)$ whenever $\langle i,j\rangle \in E$.

The bounds on $T(n)$ and $W(n)$ are obvious under the assumption that the least significant bit position in which two binary numbers differ can be found in $O(1)$ sequential time, whenever each of the two binary numbers is of size $O(\log n)$ bits. We avoid giving a justification of this assumption here, since it involves a discussion of the specific type of primitive operations allowed on each processor. □

2.7.2 A SUPERFAST 3-COLORING ALGORITHM

We start by estimating the number of colors generated by Algorithm 2.9 as a function of the number of initial colors.

Let $t > 3$ be the number of bits used to represent each of the colors in the initial coloring c. Then, each of the colors used by c' can be represented with $\lceil \log t \rceil + 1$ bits. Hence, if the number of colors in c is q, where q satisfies $2^{t-1} < q \leq 2^t$, coloring c' uses at most $2^{\lceil \log t \rceil + 1} = O(t) = O(\log q)$ colors. Therefore, the number of colors decreases exponentially in general.

The basic coloring procedure can be applied repeatedly. Further reduction of the number of colors occurs as long as the number t of bits used to represent the colors satisfies $t > \lceil \log t \rceil + 1$—that is, $t > 3$. Applying the basic coloring procedure to the case where $t = 3$ will result in a coloring with at most six colors, since the new color of i is given by $c'(i) = 2k + c(i)_k$, and hence $0 \leq c'(i) \leq 5$ (as $0 \leq k \leq 2$). We next estimate the number of iterations needed to reach this stage.

Before presenting the analysis on the number of iterations, we introduce the following notation. Let $\log^{(i)} x$ be defined by $\log^{(1)} x = \log x$, and $\log^{(i)} x = \log (\log^{(i-1)} x)$. For example, $\log^{(2)} x = \log \log n$, and $\log^{(3)} x = \log \log \log x$. Let $\log^* x = \min\{i \mid \log^{(i)} x \leq 1\}$. The function $\log^* x$ is an extremely slowly growing function that is bounded by 5 for all $x \leq 2^{65536}$.

Starting with the initial coloring $c(i) = i$ for $1 \leq i \leq n$, the first application of Algorithm 2.9 reduces the number of colors to $O(\log n)$ colors. The second application reduces the number of colors to $O(\log \log n) = O(\log^{(2)} n)$. Therefore, the number of colors will be reduced to less than or equal to six after $O(\log^* n)$ iterations.

We can reduce the number of colors to three as follows. The additional recoloring procedure consists of three iterations, each of which handles vertices of a specific color. For each $3 \leq l \leq 5$, we do the following. For each vertex i with color l, we recolor vertex i with the smallest possible color from $\{0, 1, 2\}$ (that is, smallest color different from the colors of its predecessor and its successor). Each iteration takes $O(1)$ time, using $O(n)$ operations. After the last iteration, we obtain a 3-coloring within the same asymptotic bounds.

EXAMPLE 2.14:

Let us apply the recoloring procedure to the example in Fig. 2.13. The colors of the vertices obtained after we apply Algorithm 2.9 are 0, 1, 2, 3, 4, 5. Vertex 6 is the only vertex of color 3. Since its neighbors, vertices 5 and 8, are colored 1 and 2, vertex 6 is recolored 0. Next, vertices 3 and 9 are recolored 0 and 1, respectively. Finally, vertices 13 and 14 are recolored 0 and 2, respectively. □

Theorem 2.6: *We can color the vertices of a directed cycle with three colors in $O(\log^* n)$ time, using $O(n \log^* n)$ operations.*

Proof: The procedure consists of $O(\log^* n)$ iterations of the basic coloring algorithm (Algorithm 2.9), followed by recoloring the vertices whose colors are between colors 3 and 5.

As we have seen before, the number of colors obtained after $O(\log^* n)$ iterations of Algorithm 2.9 is less than or equal to six. The recoloring procedure ensures that vertices of colors 4, 5, and 6 are recolored from the set $\{0, 1, 2\}$. The recoloring procedure works correctly because no adjacent vertices are recolored at the same time.

Obtaining the stated bounds on the resources is straightforward. □

PRAM Model: In Algorithm 2.9, simultaneous access to an entry $c(i)$ can be avoided by creating a copy of c and accessing the copy for the color of the successor. Therefore this algorithm runs on the EREW PRAM model. □

2.7.3 AN OPTIMAL 3-COLORING ALGORITHM

The previous 3-coloring algorithm in Section 2.7.2 breaks the symmetry in a directed cycle extremely fast, but uses a nonlinear number of operations. However, for all practical purposes, this algorithm takes less than six parallel steps, each of which involves simple operations on each of the vertices.

If we insist on optimality, a more involved variation of the procedure can be used. The remark that follows is needed for our optimal algorithm.

Remark 2.5: (**Sorting Integers**) Sorting n integers, each of which in the range $[0, O(\log n)]$, can be done in $O(\log n)$ time using a linear number of operations. This sorting can be accomplished by using a combination of the well-known radix-sorting algorithm and the prefix-sums algorithm. The reader is asked to provide the details in Exercise 2.45. □

Our optimal algorithm consists of (1) coloring the vertices with $O(\log n)$ colors using the basic coloring procedure, (2) sorting the vertices by their colors (hence, all vertices with the same color appear consecutively), and (3) applying a recoloring procedure successively on each set of vertices with the same color. The details are given in the following algorithm.

ALGORITHM 2.10
(3-Coloring of a Cycle)
Input: *A directed cycle of length n whose arcs are specified by an array S.*
Output: *A 3-coloring of the vertices of the cycle.*
begin
1. **for** $1 \leq i \leq n$ **pardo**
   Set $C(i) := i$
2. *Apply Algorithm 2.9 once.*
3. *Sort the vertices by their colors.*

4. **for** $i = 3$ **to** $2\lceil \log n \rceil$ **do**
   for *all vertices v of color i* **pardo**
   Color v with the smallest color from $\{0, 1, 2\}$ *that is different from the colors of its two neighbors.*

end

Theorem 2.7: *We can color the vertices of a directed cycle with three colors in* $O(\log n)$ *time, using a total of* $O(n)$ *operations.*

Proof: Note that, by Lemma 2.3 no two adjacent vertices will have the same color after the execution of step 2. Hence, no two vertices of the same color are adjacent. The recoloring performed at step 4 is therefore valid.

As for the time complexity, steps 1 and 2 take $O(1)$ time, using $O(n)$ operations, and step 3 takes $O(\log n)$ time, using $O(n)$ operations, in view of Remark 2.5. After sorting the vertices by their colors, we can assume that all the vertices with the same color are in consecutive memory locations, and that we know the locations of the first and the last vertices of each color (see Exercise 2.40). Let n_i be the number of vertices of color i. Recoloring these vertices takes $O(1)$ parallel time, using $O(n_i)$ operations. Therefore step 4 takes $O(\log n)$ time, using a total of $O(\sum_i n_i) = O(n)$ operations. □

PRAM Model: Algorithm 2.10 does not need any simultaneous memory access. Hence, it runs on the EREW PRAM model. □

2.7.4 REVIEW

A simple technique was presented in this section to perform a 3-coloring of the vertices of a cycle. This technique seems to break the symmetry present in some problems. Examples of the applications of this technique are given in the exercises at the end of this chapter, as well as in later chapters. Another technique for breaking symmetry uses randomization, a topic studied in Chapter 9.

2.8 Summary

In this chapter, we provided a collection of general methods for designing parallel algorithms. These methods are adapted for solving many of the

problems encountered as we progress in this book. Our classification is somewhat arbitrary, in the sense that a given algorithm could sometimes be grouped easily under two or more of these methods. For example, the $O(\log n)$ time parallel algorithm for computing the sum of n numbers also could have been described as a divide-and-conquer algorithm.

The parallel algorithms developed for the specific combinatorial problems are commonly used as building blocks to solve more complex problems. Table 2.1 gives a summary of the most important algorithms.

TABLE 2.1
ALGORITHMS INTRODUCED IN THIS CHAPTER.

Algorithm		Section	Time	Work	Method	PRAM Model
2.1	Prefix Sums	2.1.1	$O(\log n)$	$O(n)$	Balanced binary tree	EREW
2.2	Nonrecursive Prefix Sums	2.1.2	$O(\log n)$	$O(n)$	Balanced binary tree	EREW
2.4	Pointer Jumping	2.2.1	$O(\log h)$	$O(n \log h)$	Pointer jumping	CREW
2.5	Parallel Prefix (Rooted Trees)	2.2.2	$O(\log h)$	$O(n \log h)$	Pointer jumping	CREW
2.6	Upper Hull	2.3.2	$O(\log^2 n)$	$O(n \log n)$	Divide and conquer	CREW
2.7	Partition	2.4.2	$O(\log n)$	$O(n + m)$	Partitioning	CREW
	Merging	2.4.2	$O(\log n)$	$O(n)$	Partitioning	CREW
2.8	Basic Maximum	2.6.1	$O(1)$	$O(n^2)$	All pairs comparisons	Common CRCW
	Fast, Optimal Maximum	2.6.2 and 2.6.3	$O(\log \log n)$	$O(n)$	Accelerated cascading	Common CRCW
2.9	Basic Coloring	2.7.1	$O(1)$	$O(n)$	Symmetry breaking	EREW
	Fast Coloring	2.7.2	$O(\log^* n)$	$O(n \log^* n)$	Symmetry breaking	EREW
2.10	3-Coloring of a Cycle	2.7.3	$O(\log n)$	$O(n)$	Symmetry breaking and integer sort	EREW

Exercises

2.1. Develop an optimal nonrecursive prefix-sums algorithm that is similar to Algorithm 2.2 but that does not use the auxiliary variables B and C. The input array A should hold the prefix sums when the algorithm terminates.

2.2. Consider the following divide-and-conquer algorithm for computing the prefix sums $\{s_i\}$ of a sequence $x_1, x_2, \ldots, x_n$, where n is a power of 2. Compute the prefix sums of the two subsequences $\{x_1, \ldots, x_{n/2}\}$ and $\{x_{(n/2)+1}, \ldots, x_n\}$—say, $z_1, \ldots, z_{n/2}$ and $z_{(n/2)+1}, \ldots, z_n$, respectively. Then, set $s_i = z_i$, for $1 \le i \le n/2$, and $s_i = z_i + z_{n/2}$, for $(n/2) + 1 \le i \le n$.

Write a nonrecursive version of this algorithm in the WT presentation framework. What are the time and the work required by the algorithm? What is the PRAM model needed?

2.3. Suppose we are given a set of n elements stored in an array A together with an array L such that $L(i) \in \{1, 2, \ldots, k\}$ represents the label of element $A(i)$, where k is a constant. Develop an optimal $O(\log n)$ time EREW PRAM algorithm that stores all the elements of A with label 1 into the upper part of A while preserving their initial ordering, followed by the elements labeled 2 with the same initial ordering, and so on.

2.4. Let $A = (a_1, a_2, \ldots, a_n)$ be an array of elements, and let $j_1 = 1 < j_2 < \cdots < j_s = n$ be a set of indices. Consider the problem of computing the array $B = (b_1, b_2, \ldots, b_n)$ such that $b_l = a_{j_i}$, for $j_{i-1} < l \le j_i$ and $2 \le i \le s$. For example, the array B corresponding to $A = (4, 7, 9, 6, 15)$ and $j_1 = 1 < j_2 = 3 < j_3 = 5$ is given by $B = (9, 9, 9, 15, 15)$. Develop an optimal algorithm whose running time is $O(\log n)$.

2.5. (**Segmented Prefix Sums**) We are given a sequence $A = (a_1, a_2, \ldots, a_n)$ of elements from a set S with an associative operation $*$, and a Boolean array B of length n such that $b_1 = b_n = 1$. For each $i_1 < i_2$ such that $b_{i_1} = b_{i_2} = 1$ and $b_j = 0$ for all $i_1 < j < i_2$, we wish to compute the prefix sums of the subarray $(a_{i_1+1}, \ldots, a_{i_2})$ of A. Develop an $O(\log n)$ time algorithm to compute all the corresponding prefix sums. Your algorithm should use $O(n)$ operations and should run on the EREW PRAM.

2.6. Let $A = (a_1, a_2, \ldots, a_n)$ be an array of elements drawn from a linearly ordered set. The *suffix-minima problem* is to compute for each i, where $1 \le i \le n$, the minimum element among $\{a_i, a_{i+1}, \ldots, a_n\}$. We can, in a similar fashion, define the *prefix minima*. Develop an $O(\log n)$ time algorithm to compute the prefix and the suffix minima of A using a total of $O(n)$ operations. Your algorithm should run on the EREW PRAM.

2.7. Given an array $A = (a_1, \ldots, a_n)$, where the elements come from a linearly ordered set S, and given two elements $x, y \in S$, show how to store all the elements a_i from A that are between x and y in consecutive memory locations. Your algorithm should run in $O(\log n)$ time using $O(n)$ operations. Specify the PRAM model used.

2.8. Let x be an indeterminate and let n be a positive integer. Consider the problem of computing the polynomials $y_i = x^i$, for $1 \leq i \leq n$. Show how to compute all the y_i's in $O(\log n)$ time using $O(n)$ operations.

2.9. Given n independent jobs with processing times $\{t_1, t_2, \ldots, t_n\}$, determine a schedule of each job on a set of m machines such that the finish time is minimum. A job can be split to run on different machines. Your algorithm should run in $O(\log n)$ time using a total of $O(n)$ operations.

2.10. Consider a cycle $C = (v_1, v_2, \cdots, v_n)$ with an additional set E of edges between the vertices of C such that, for each vertex v_i, there exists at most one edge in E incident on v_i. Consider the problem of determining whether or not it is possible to draw all the edges in E inside the cycle C without any two of them crossing. Develop an $O(\log n)$ time algorithm to solve this problem. The total number of operations used must be $O(n)$. Your algorithm should run on the EREW PRAM model.

2.11. Develop an $O(\log n \log m)$ time algorithm to compute A^i, for all $1 \leq i \leq m$, where A is an $n \times n$ matrix. What is the amount of work required?

2.12. Let F be a forest of rooted directed trees specified by an array S of length n such that $S(i) = j$ if and only if (i, j) is an arc in F. Show that any algorithm to determine the root of the tree containing node i, for $1 \leq i \leq n$, will take $\Omega(\log n)$ time on the EREW PRAM even if the maximum height of any tree is very small.

2.13. Let A be a Boolean array of size n.

a. Develop an $O(1)$ time CRCW PRAM algorithm to find the smallest index k such that $A(k) = 1$. The total number of operations must be $O(n)$. *Hint*: Use a $\sqrt{n}$ divide-and-conquer strategy. Start by solving the subproblems.

b. How fast can you solve this problem on the CREW PRAM? Your algorithm must use $O(n)$ operations.

2.14. Given an array $A = (a_1, a_2, \ldots, a_n)$ whose elements are drawn from a linearly ordered set, the *left match* of a_i, where $1 \leq i \leq n$, is the element a_k, if it exists, such that k is the maximum index satisfying $1 \leq k < i$ and $a_k < a_i$. Similarly, we can define the right match of a_i. The problem of finding the right and left matches of all the elements in A is called the problem of *all nearest smaller values* (ANSV).

Show how to solve the ANSV problem in $O(1)$ time using $O(n^2)$ operations. *Hint*: Use Exercise 2.13.

2.15. *Given a set of n open intervals (x_i, y_i) with $x_i < y_i$, where $1 \leq i \leq n$, consider the problem of packing these intervals on a minimum number of horizontal lines such that no two intervals on the same horizontal line overlap. Assume that the $2n$ points $\{x_i, y_i\}_{i=1}^{n}$ are sorted in non-decreasing order, with ties broken according to the index i. Develop an $O(\log n)$ time algorithm to achieve this. What is the total number of operations used? *Hint*: Adjust the sequence of the $2n$ points and, for each y_i, determine a closest $x_j > y_i$.

2.16. Show that the problem of finding the ordered list of vertices defining the convex hull of n points in the plane requires $\Omega(n \log n)$ operations. *Hint*: Consider the set of points (x_i, x_i^2), where $1 \leq i \leq n$.

2.17. Show how to modify Algorithm 2.6 such that it runs on the EREW PRAM model without asymptotically increasing the bounds on $T(n)$ and $W(n)$.

2.18. * An optimal $O(\log n)$ time algorithm to solve the convex-hull problem can be developed with the following divide-and-conquer strategy.

Let S be the set of points. Partition S into $S_1, S_2, \ldots, S_{\sqrt{n}}$, each of size approximately $\sqrt{n}$, by sorting the points by their x coordinates. Compute $CH(S_i)$ recursively, and then combine all the $CH(S_i)$ into a solution.

a. Let $T_{i,j}$ be the upper common tangent between $CH(S_i)$ and $CH(S_j)$. Develop an algorithm that determines which points of $CH(S_i)$, if any, belong to $CH(S)$. Your algorithm should run in $O(\log n)$ time using $O(\sqrt{n} \log n)$ operations. *Hint*: Let L_i be the tangent of smallest slope in $\{T_{i,1}, T_{i,2}, \ldots, T_{i,i-1}\}$, and let M_i be the tangent of largest slope in $\{T_{i,i+1}, T_{i,i+2}, \ldots, T_{i,\sqrt{n}}\}$.

b. Develop the recurrence equations for the running time $T(n)$ and the total number of operations $W(n)$ required by the overall algorithm. Show that $T(n) = O(\log n)$ and $W(n) = O(n \log n)$.

2.19. We have already introduced, in Exercise 2.6, the problem of computing the prefix and the suffix minima of an array A of n elements whose elements are drawn from a linearly ordered set.

a. Use Exercise 2.14 to design an $O(1)$ time algorithm for computing the prefix and the suffix minima of A, using a total of $O(n^2)$ operations.

b. *Use a $\sqrt{n}$ divide-and-conquer strategy to obtain an $O(\log \log n)$ time algorithm. The total number of operations used must be $O(n)$. Specify the PRAM model needed.

2.20. Let A be an $n \times n$ lower triangular matrix such that $n = 2^k$. Assume that A is nonsingular. Let A be partitioned into blocks, each of size $\frac{n}{2} \times \frac{n}{2}$, as follows:

$$\begin{bmatrix} A_{11} & 0 \\ A_{21} & A_{22} \end{bmatrix}$$

a. Show that the inverse of A is given by the following:

$$\begin{bmatrix} A_{11}^{-1} & 0 \\ -A_{22}^{-1}A_{21}A_{11}^{-1} & A_{22}^{-1} \end{bmatrix}$$

b. Develop a divide-and-conquer algorithm to compute A^{-1} in $O(\log^2 n)$ time. What is the total number of operations used? Assume that the addition, multiplication, and division of two real numbers are primitive operations on your PRAM.

2.21. Let A and B be two sorted sequences whose elements are not necessarily distinct. Show how to use Algorithm 2.7 to merge A and B without increasing the resources required.

2.22. Develop an algorithm using the partitioning strategy outlined in Section 2.4 to merge two sequences of lengths n and m, where $n \neq m$. What is the running time of your algorithm? What is the total number of operations?

2.23. Sketch a solution to the processor allocation problem required to implement Algorithm 2.7 on the CREW PRAM model.

2.24. Let A be an arbitrary array whose n elements are drawn from a linearly ordered set S, and let $x \in S$.

a. Show how to determine $rank(x : A)$ in $O(\log n)$ time using $O(n)$ operations. Your algorithm should run on the EREW PRAM model.

b. Suppose A is sorted. How fast can you determine $rank(x : A)$ with $o(n)$ operations? Specify the PRAM model used.

2.25. Let A and B be two sorted arrays, each of length n. Suppose that we are given the integer array $rank(A : B)$. Develop an optimal $O(\log n)$ time algorithm to compute $rank(B : A)$. Your algorithm should run on the EREW PRAM model.

2.26. Let $B = (b_1, b_2, \ldots, b_k)$ be a sequence of elements stored in the global memory of an n-processor EREW PRAM.

a. Develop an algorithm to distribute a copy of B to each of the local memories of the n processors in $O(\log n)$ time, assuming that $k = O(\log n)$.

b. Generalize your algorithm to handle t such sequences that must be distributed to all the processors.

2.27. Let T be a 2-3 tree of height h. Show that the number of vertices of T is between $2^{h+1} - 1$ and $(3^{h+1} - 1)/2$, and that the number of leaves is between 2^h and 3^h.

2.28. Let T be a 2-3 tree with n leaves. You wish to search for a given set of k elements in T, where $k < n$. Show how to accomplish this search in $O(\log n)$ time on the EREW PRAM model.

2.29. Show how to delete a given sorted sequence of k elements from a 2-3 tree with n leaves in $O(\log n)$ time, using a total of $O(k \log n)$ operations. Use the CREW PRAM model if you wish.

2.30. Let A be an arbitrary array of size n, and let B be a nondecreasing array of size n whose elements are drawn from $\{1, 2, \ldots, n\}$.

a. Develop an $O(\log n)$ time EREW PRAM algorithm to compute the array C such that $c_i = a_{b_i}$, for $1 \leq i \leq n$. You must use a linear number of operations.

b. Suppose that each a_i of A is replaced with a linear block of size $\lceil \log n \rceil$. Show how to generate the corresponding $n \times \lceil \log n \rceil$ array C in $O(\log n)$ time optimally on the EREW PRAM.

2.31. Solve the processor allocation problem corresponding to the algorithm given in Section 2.5 to insert a sequence of k elements in a 2-3 tree with n leaves.

2.32. Develop an algorithm that, given a sorted array A of n elements, constructs a 2-3 tree T to represent A.

a. Specify T so as to make the correspondence between A and T one to one.

b. Show how to build such a tree optimally on the CREW PRAM model. What is the running time of your algorithm?

2.33. Let T be a 2-3 tree with n leaves holding the items $a_1 < a_2 < \ldots < a_n$. Suppose you wish to insert k elements, $b_1 < b_2 < \ldots < b_k$. One approach would be to build a new 2-3 tree with the corresponding $n + k$ leaves without using the given tree T at all. Develop such an algorithm and state its running time and the total number of operations used. Compare with the insertion algorithm described in Section 2.5.2.

2.34. Let $T_0, T_1, \cdots, T_k$ be 2-3 trees such that, for any $0 \leq i < j \leq k$, if a_i is a leaf stored in T_i and a_j is a leaf stored in T_j, then $a_i < a_j$. Develop an $O(\log n + \log k)$ time algorithm to combine $T_0, T_1, \cdots, T_k$ into a new 2-3 tree. Your algorithm should not use any simultaneous memory access.

2.35. Develop an algorithm similar to Algorithm 2.2 to compute the maximum of n elements using a doubly logarithmic-depth tree. Assume that $n = 2^{2^k}$, for some positive integer k.

2.36. Show how to compute the maximum of n elements in $O(1)$ parallel time using n^{1+c} processors, where c is an arbitrary positive constant. Which PRAM model do you need?

2.37. Suppose that we have an algorithm A to solve a given problem P of size n in $O(\log n)$ time on the PRAM using $O(n \log n)$ operations. On the other hand, an algorithm B exists that reduces the size of P by a constant fraction in $O(\log n/\log \log n)$ time using $O(n)$ operations without altering the solution. Derive an $O(\log n)$ time algorithm to solve P using $O(n)$ operations.

2.38. You are given an array of colors $A = (a_1, a_2, \ldots, a_n)$ drawn from k colors $\{c_1, c_2, \ldots, c_k\}$, where k is a constant. You wish to compute k

indices $i_1, i_2, \ldots, i_k$, for each element a_i, such that i_j is the index of the closest element to the left of a_i whose color is c_j. If no such element exists, then set $i_j = 0$. Show how to solve this problem in $O(\log n)$ time using a total of $O(n)$ operations on the EREW PRAM.

2.39. You are given a sorted array $A = (a_1, a_2, \ldots, a_n)$ such that each a_i is labeled l_i, where $l_i \in \{1, 2, \ldots, m\}$, $m = O(\log n)$.

a. Develop an $O(\log n)$ time CREW PRAM algorithm to determine, for each $1 \leq i \leq m$, the minimum element of A with label l_i. Your algorithm must use $O(n)$ operations. Do not use any sorting algorithms.

b. What if $m = O(\log \log n)$? Which PRAM model do you need?

2.40. Develop a routine to get pointers to the first vertex and to the last vertex of the same color after step 3 of Algorithm 2.9. Your procedure should run in $O(\log n)$ time using a total of $O(n)$ operations on the EREW PRAM model.

2.41. *Let $T = (V, E)$ be a rooted tree with $V = \{1, 2, \ldots, n\}$, specified by the pairs $(i, p(i))$, $1 \leq i \leq n$, where $p(i)$ is the parent of i. If r is the root, then $p(r) = 0$.

a. Develop a CREW PRAM algorithm to color T with three colors in $O(\log^* n)$ using $O(n \log^* n)$ operations. Can you derive an optimal algorithm?

b. A *pseudoforest* is a directed graph in which each vertex has an out-degree ≤ 1. Generalize your tree-coloring algorithm from part (a) to handle pseudo forests.

2.42. Given a pseudoforest (as defined in Exercise 2.41), you wish to assign to each vertex v a label $l(v)$ such that all the vertices in the same weakly connected component (that is, a connected component in the undirected version of the graph) get the same label. Develop an $O(\log n)$ time algorithm to accomplish this task. What is the total number of operations required by your algorithm?

2.43. Let $T = (V, E)$ be a rooted tree specified by the pairs $(i, p(i))$, $1 \leq i \leq n$, where $p(i)$ is the parent of i. Show that T can always be 2-colored. Develop an $O(\log n)$ time CREW PRAM algorithm to find such a coloring.

2.44. *Given a set of n integers, each colored with one color from $\{1, 2, \ldots, k\}$, where $k \leq \log n$, show how to obtain the prefix sums of the elements of a given color i, in the order they appear, for all i. Your algorithm should run in $O(\log n)$ time using a total of $O(n)$ operations on the EREW PRAM.

2.45. *Suppose we are given n integers in the range $[0, \log n - 1]$. Develop an $O(\log n)$ time EREW PRAM sorting algorithm that uses $O(n)$ operations. Can you generalize your algorithm to the case when the range is

$[0, m], m \geq \log n$? What is the corresponding time? *Hint*: Use radix sort, and start by computing the number of elements equal to i, for each i. Use your answer to Exercise 2.44.

2.46. *Given an undirected graph $G = (V, E)$ such that $|V| = n$ and the maximum degree of any vertex v is bounded by a constant δ, show how to color G with $\delta + 1$ colors in $O(\log^* n)$ time using $O(n \log^* n)$ operations. *Hint*: Decompose the graph into $O(\delta)$ pseudoforests. Color each pseudoforest with three colors as in Exercise 2.41, and recolor G appropriately.

Bibliographic Notes

The parallel algorithm for computing the prefix sums was developed by Ladner and Fischer [13]. The design of efficient parallel algorithms based on balanced binary trees was emphasized in [6], where this technique was applied to several scheduling problems. The pointer jumping technique was used in several of the early parallel graph algorithms; for example, see [10, 16, 11]. Wyllie [21] used this technique on linked lists to solve the parallel prefix problem. The divide-and-conquer approach for computing the planar convex hull was originally proposed by Shamos [17] for its sequential computation. Valiant [19] gave the first (nonoptimal) $O(\log \log n)$ time algorithm for computing the maximum of n elements, and introduced the partitioning strategy to solve the merging problem on the parallel comparison-tree model (to be introduced in Chapter 4). The optimal $O(\log \log n)$ time algorithm to compute the maximum, and an optimal $O(\log n)$ time merging algorithm, were given in [18]. The accelerated-cascading technique was introduced in [4]. Pipelining is a classical technique that can be traced back to the early days of computing. Our algorithm to insert a sequence of elements into a 2-3 tree is taken from [14]. The technique to break the symmetry in coloring the vertices of a cycle was introduced by Cole and Vishkin in [5], and was later simplified in [8]. A similar technique was developed independently by Wagner and Han [20]. The solutions to Exercises 2.13 and 2.15 can be found in [7] and [12], respectively. Exercises 2.14 and 2.19 are taken from [3]. The divide-and-conquer strategy sketched in Exercise 2.20 was given in [9]. The $\sqrt{n}$ divide-and-conquer strategy to solve the convex-hull problem described in Exercise 2.18 was developed independently in [1] and [2]. Exercises 2.28 and 2.29 are taken from [14]. Solutions to Exercises 2.41 and 2.46 can be found in [8], and the solutions to Exercises 2.44 and 2.45 appear in [5, 15].

References

1. Aggarwal, A., B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap. Parallel computational geometry. *Algorithmica*, 3(3):293–327, 1988.
2. Atallah, M. J., and M. T. Goodrich. Efficient parallel solutions to some geometric problems. *Journal of Parallel and Distributed Computing*, 3(4):492–507, 1986.
3. Berkman, O., B. Schieber, and U. Vishkin. Some doubly logarithmic optimal parallel algorithms based on finding all nearest smaller values. Technical Report UMIACS-TR-88-79, Institute for Advanced Computer Studies, University of Maryland, College Park, MD, 1988.

4. Cole, R., and U. Vishkin. Approximate coin tossing with applications to list, tree and graph problems. In *Proceedings Twenty-Seventh Annual IEEE Symposium on Foundations of Computer Science*, Toronto, Canada, 1986, pages 478–491. IEEE Press, Piscataway, NJ.
5. Cole, R., and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986.
6. Dekel, E., and S. Sahni. Parallel scheduling algorithms. *Operations Research*, 31(1):24–49, 1983.
7. Fich, F. E., P. Ragde, and A. Wigderson. Relations between concurrent-write models of parallel computation. *SIAM J. Computing*, 17(3):606–627, 1988.
8. Goldberg, A., S. Plotkin, and G. Shannon. Parallel symmetry-breaking in sparse graphs. In *Proceedings Nineteenth Annual ACM Symposium on Theory of Computing*, New York, NY, 1987, pages 315–324.
9. Heller, D. A survey of parallel algorithms in numerical linear algebra. *SIAM Review*, 20(4):740–777, 1978.
10. Hirschberg, D. S. Parallel algorithms for the transitive closure and the connected components problems. In *Proceedings Eighth Annual ACM Symposium on Theory of Computing*, Hershey, PA, 1976, pages 55–57. ACM Press, New York.
11. JáJá, J. Graph connectivity problems on parallel computers. Technical Report CS-78-05, Department of Computer Science, Pennsylvania State University, University Park, PA, 1978.
12. JáJá, J., and S-C. Chang. Parallel algorithms for channel routing in the knock-knee model. *SIAM J. Computing*, 20 (2):228–245, 1991.
13. Ladner, R. E., and M. J. Fischer. Parallel prefix computation. *JACM*, 27(4):831–838, 1980.
14. Paul, W., U. Vishkin, and H. Wagener. Parallel dictionaries on 2-3 trees. In *Proceedings Tenth ICALP*, Barcelona, Spain, 1983, Volume 154, pages 597–609. Springer-Verlag.
15. Rajasekaran, S., and J. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Computing*, 18(3):594–607, 1989.
16. Savage, C. *Parallel Algorithms for Graph Theoretic Problems*. PhD thesis, Computer Science Department, University of Illinois, Urbana, IL, 1978.
17. Shamos, M. I. *Computational Geometry*. PhD thesis, Department of Computer Science, Yale University, New Haven, CT, 1978.
18. Shiloach, Y., and U. Vishkin. Finding the maximum, merging and sorting in a parallel computation model. *Journal of Algorithms*, 2(1):88–102, 1981.
19. Valiant, L. G. Parallelism in comparison problems. *SIAM J. Computing*, 4(3):348–355, 1975.
20. Wagner, W., and Y. Han. Parallel algorithms for bucket sorting and the data dependent prefix problem. In *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL, 1986, pages 924–930.
21. Wyllie, J. C. *The Complexity of Parallel Computations*. PhD thesis, Computer Science Department, Cornell University, Ithaca, NY, 1979.

3

Lists and Trees



Lists and **trees** are basic data structures that are used frequently in many processing tasks. Extensive literature exists on lists and trees, their properties, and their uses. This chapter addresses several basic problems in processing lists and in computing tree functions.

One of the most elementary list-processing tasks is to rank the nodes from either end of the list, a problem that we call the *list-ranking problem*. A linear-time sequential algorithm can be easily developed to handle this problem, but the design of a fast and optimal parallel algorithm turns out to be significantly more challenging. In Section 3.1, we present two parallel algorithms to solve the list-ranking problem, each using a linear number of operations. The first is a simple algorithm that runs in $O(\log n \log\log n)$ time; the second, whose analysis is considerably more involved, runs in $O(\log n)$ time. Each algorithm uses an interesting combination of several techniques.

As for trees, we introduce two powerful techniques: the *Euler tour* (Section 3.2) and *tree contraction* (Section 3.3). These techniques are then used for the computation of tree functions, for *arithmetic expression evaluation*, and for solving the *lowest common ancestor problem*.

3.1 List Ranking

Consider a linked list L of n nodes whose order is specified by an array S such that $S(i)$ contains a pointer to the node following node i on L, for $1 \leq i \leq n$. We assume that $S(i) = 0$ when i is the end of the list. The data stored in the nodes play no role in this section. The **list-ranking problem** is to determine the distance of each node i from the end of the list.

The list-ranking problem is one of the most elementary problems in list processing whose sequential complexity is trivially linear. It also arises in other contexts, such as computing tree functions and solving graph-theoretic problems.

The pointer jumping technique (Algorithm 2.5) can be used to derive a parallel algorithm for the list-ranking problem. The corresponding running time is $O(\log n)$, and the corresponding total number of operations is $O(n \log n)$. This algorithm is nonoptimal in view of the existence of a linear-time sequential algorithm.

The pointer jumping algorithm can be made optimal if we can somehow reduce the size of the list to $O(n/\log n)$ nodes using a linear number of operations. The standard strategy to achieve optimality (recall Remark 2.4) would be (1) to partition the input list into approximately $n/\log n$ blocks $\{B_i\}$, each containing $O(\log n)$ nodes; (2) to rank each node within its block (called the *preliminary rank*) by using an optimal sequential algorithm; and (3) to combine the preliminary ranks using an $O(\log n)$ time parallel algorithm. Unfortunately, each block can have $\Omega(\log n)$ sublists, in which case the size of the input list to the $O(\log n)$ time parallel algorithm would not necessarily have been reduced to $O(n/\log n)$ nodes. Therefore, we need an alternative method. The remainder of this section covers two strategies to shrink the given list efficiently.

3.1.1 A SIMPLE OPTIMAL LIST-RANKING ALGORITHM

Given a linked list L with n nodes, we would like to compute an array R such that $R(i)$ is equal to the distance of node i from the end of L. Initially, we set $R(i) = 1$ for all nodes i, except for the last node, whose R value is set to 0.

We start by stating the parallel algorithm based on the pointer jumping technique. This algorithm is essentially the same as Algorithm 2.5, which was given for rooted directed trees.

ALGORITHM 3.1
(List Ranking Using Pointer Jumping)
Input: *A linked list of n nodes such that (1) the successor of each node i is given by S(i), and (2) the S value of the last node on the list is equal to* 0.

Output: *For each* $1 \leq i \leq n$, *the distance* $R(i)$ *of node* i *from the end of the list.*

begin

1. **for** $1 \leq i \leq n$ **pardo**
  if $(S(i) \neq 0)$ **then** *set* $R(i) := 1$
    else *set* $R(i) := 0$

2. **for** $1 \leq i \leq n$ **pardo**
  Set $Q(i) := S(i)$
  while $(Q(i) \neq 0$ *and* $Q(Q(i)) \neq 0)$ **do**
    Set $R(i) := R(i) + R(Q(i))$
    Set $Q(i) := Q(Q(i))$

end

Since this algorithm was examined before, we leave the proof of the following lemma to Exercise 3.1.

Lemma 3.1: *Given a linked list of n nodes, Algorithm 3.1 generates the rank* $R(i)$ *of each node i in* $O(\log n)$ *time using* $O(n \log n)$ *operations. Moreover, the algorithm can be implemented to run on the EREW PRAM.* □

The overall strategy for solving the list-ranking problem optimally is outlined next.

List-Ranking Strategy

1. Shrink the linked list L until only $O(n/\log n)$ nodes remain.
2. Apply the pointer jumping technique (Algorithm 3.1) on the short list of the remaining nodes.
3. Restore the original list and rank all the nodes removed in step 1.

We have just examined how to implement step 2 of this strategy. Since the number of elements is $O(n/\log n)$, Algorithm 3.1 takes $O(\log n)$ time using $O(n)$ operations. Step 3 consists of reversing the process of step 1, and its details depend on the implementation of step 1. A possible scheme is outlined later. The main difficulty lies in performing step 1 in $O(\log n)$ time using a linear number of operations.

Independent Sets. The method for shrinking the list L consists of removing a selected set of nodes from L and updating the intermediate R values of the remaining nodes. The key to a fast parallel implementation lies in using an *independent* set of nodes to be removed. A set I of nodes is **independent** if, whenever $i \in I$, $S(i) \notin I$. We can remove each node $i \in I$ by adjusting the successor pointer of the predecessor of i. Since I is independent, this process can be applied concurrently to all the nodes in I.

The information concerning the removed nodes should be stored somewhere so that later the original list can be restored and the nodes in I can be ranked properly. In the following procedure, we use the array U to store such information. A separate array is needed because our list contraction procedure (Algorithm 3.3) will compact the remaining elements into consecutive memory locations. Without loss of generality, we assume that the *predecessor* array P is available. Otherwise, it can be set up in $O(1)$ time using $O(n)$ operations.

ALGORITHM 3.2
(Removing Nodes of an Independent Set)
Input: *(1) Arrays S and P of length n representing, respectively, the successor and the predecessor relations of a linked list; (2) an independent set I of nodes such that* $P(i), S(i) \neq 0$*; (3) a value* $R(i)$ *for each node i.*
Output: *The list obtained after removal of all the nodes in I with the updated R values.*
begin
1. Assign consecutive serial numbers $N(i)$ *to the elements of I, where* $1 \leq N(i) \leq |I| = n'$.
2. **for** *for all* $i \in I$ **pardo**
Set $U(N(i)) := (i, S(i), R(i))$
Set $R(P(i)) := R(P(i)) + R(i)$
Set $S(P(i)) := S(i)$
Set $P(S(i)) := P(i)$
end

Lemma 3.2: *Given a linked list L of size n and an independent set I, Algorithm 3.2 correctly removes the nodes of I and updates the R values in* $O(\log n)$ *time using* $O(n)$ *operations.*

Proof: The correctness proof follows from the fact that no two nodes of I are adjacent. As for the running time, step 1 takes $O(\log n)$ time using $O(n)$ operations by a prefix-sums computation on the nodes of L such that a weight of 1 is assigned to each node in I, and a weight of 0 is assigned to each of the remaining nodes. Step 2 can be executed in $O(1)$ time, using $O(n)$ operations. □

Once the ranks of the nodes in the contracted list are determined, it is easy to obtain the ranks of the deleted nodes and to restore the original list using the information stored in the U array.

EXAMPLE 3.1:

Figure 3.1 illustrates the process of contracting a list for a given independent set. The initial list and the initial R values (in brackets) are shown in Fig. 3.1(a).

An independent set of nodes consists of $\{1, 5, 6\}$. Step 1 assigns to the nodes $\{1, 5, 6\}$ the serial numbers $N(1) = 1, N(5) = 2$, and $N(6) = 3$. Executing the loop of step 2 for node 6, we obtain $U(N(6)) = U(3) = (6, 8, 2), R(2) = 4, S(2) = 8$, and $P(8) = 2$. We proceed similarly for the nodes 5 and 1. The new links and the new R values of the contracted list are shown in Fig. 3.1(b). Suppose we now have the ranks of all the nodes on the short list considered with the updated R values. Since $U(3) = (6, 8, 2)$, we conclude that node 8 is the initial successor of node 6, and thus $R(6) = R(8) + 2 = 3$. Moreover, we can reinsert this node by setting $P(6) = P(8) = 2, S(P(6)) = S(2) = 6$, $P(8) = 6$. □

Remark 3.1: Suppose we want to view Algorithm 3.2 as being executed by a set of p processors, where $p \leq |I|$. Then, the n' elements of I will be divided almost evenly into p blocks, each of which will be handled separately by a processor. Each processor can push on a private stack the information concerning the nodes it removes. Restoring the list will be accomplished by popping each such stack. □

Remark 3.2: If, after removal of the nodes of the independent set I, the remaining nodes do not have to be relocated, then step 1 of Algorithm 3.2 is unnecessary; hence, the resulting algorithm runs in $O(1)$ time, using $O(n)$ operations. □

Determination of an Independent Set. We can handle the problem of finding an independent set by coloring the nodes of the list L. Recall that a

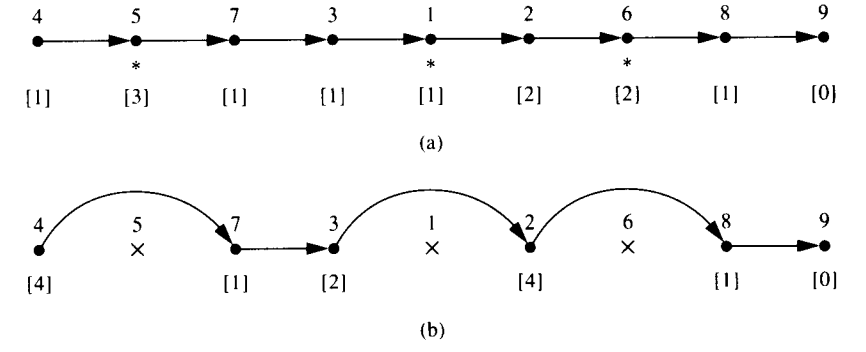


FIGURE 3.1
The process of contracting a list. (a) The initial list. The R values are given in brackets below the nodes, and the selected nodes to be removed are labeled with stars *. (b) The resulting list after contraction.

k-coloring of L is a mapping from the set of nodes in L into $\{0, 1, \ldots, k-1\}$ such that no adjacent vertices are assigned the same color (Section 2.7). A k-coloring of L can be used to determine an independent set of L as follows. A node u is a **local minimum (maximum)** with respect to this coloring if the color of u is smaller (larger) than the colors of its predecessor and its successor.

Lemma 3.3: *Given a k-coloring of the nodes of a list L of size n, the set I of local minima (or maxima) is an independent set of size $\Omega(n/k)$. The set I can be identified in $O(1)$ time, using a linear number of operations.*

Proof: Let u and v be two local minima such that no other local minimum exists between u and v. Clearly, nodes u and v cannot be adjacent. In addition, the colors of the nodes between u and v form an increasing sequence followed by a decreasing sequence. Hence the maximum number of nodes between u and v is equal to $2k - 3$. It follows that the set of local minima forms an independent set of size $\Omega(n/k)$. Similar arguments hold for the set of local maxima.

As for the complexity bounds, we can determine whether a node is a local minimum by comparing its color with the colors of its predecessor and its successor. Therefore, the set I can be identified in $O(1)$ time, using a linear number of operations. □

We can obtain a large independent set by using the optimal algorithm to 3-color the vertices of a cycle given in Section 2.7 (Algorithm 2.10). Using Lemma 3.3, we see that the corresponding independent set is of size greater than or equal to cn, for some constant $0 < c < 1$ (more precisely, we can choose $c = \frac{1}{5}$). Contracting this independent set reduces the size of the list by a constant factor; hence, this process can be repeated $\alpha \lceil \log \log n \rceil$ times to produce a list of size less than or equal to $n/\log n$, for some $\alpha > 0$.

Simple List-Ranking Algorithm. We are ready to give a description of simple optimal list-ranking algorithm.

ALGORITHM 3.3
(Simple Optimal List Ranking)
Input: *A linked list with n nodes such that the successor of each node i is given by $S(i)$.*
Output: *For each node i, the distance of i from the end of the list.*
begin
1. Set $n_0 := n, k := 0$
2. **while** $n_k > n/\log n$ **do**
2.1. Set $k := k + 1$.

2.2. Color the list with three colors, and identify the set I of local minima.
2.3. Remove the nodes in I, and store the appropriate information regarding the removed nodes (Algorithm 3.2).
2.4. Let n_k be the size of the remaining list. Compact the list into consecutive memory locations.
3. Apply the pointer jumping technique (Algorithm 3.1) to the resulting list.
4. Restore the original list and rank all the removed nodes by reversing the process performed in step 2.
end

EXAMPLE 3.2:

Consider the list of eight nodes shown in Fig. 3.2(a), where the weight of each node is shown in brackets under the node. During the first iteration of the **while** loop, we identify the independent set $I = \{3, 4, 2, 5\}$ derived from the 3-coloring shown in parentheses. Removing the nodes in I from the original list results in the new list shown in Fig. 3.2(b) with the adjusted weights. Note that the information concerning the removed nodes can be stored in triplets as described before, with an additional time stamp indicating the iteration number. For example, the information related to node 4 can be stored as $U(1, N(4)) = (4, S(4), R(4)) = (4, 1, 1)$, where the first parameter (in this case, 1) of U indicates the iteration number and the second (in this case, $N(4)$) indicates the serial number assigned to the node being removed.

During the second iteration, we obtain the list shown in Fig. 3.2(c). Since $n_2 = 2$, we go to step 3 and obtain $R(6) = R(6) + R(7) = 7$ and $R(7) = 3$.

The restoration of the original list proceeds in two iterations. After the first iteration (corresponding to the second iteration of the **while** loop), we obtain the list shown in Fig. 3.2(d). It is clear that the second iteration will fully restore the original list with the correct ranks assigned to all the nodes. □

Theorem 3.1: *Algorithm 3.3 ranks a linked list L with n nodes in* $O(\log n \log \log n)$ *time, using a linear number of operations.*

Proof: The correctness proof depends on the validity of the contraction process performed at step 2. Each iteration of the corresponding **while** loop consists of (1) determining a 3-coloring of the current list, (2) identifying the set I of local minima, (3) removing the nodes of I, and (4) compacting the remaining list. We have already developed algorithms to handle each of substeps 2.1 through 2.4; hence, we can assume that each of these substeps is carried out correctly. The whole iteration works correctly because I is an independent set.

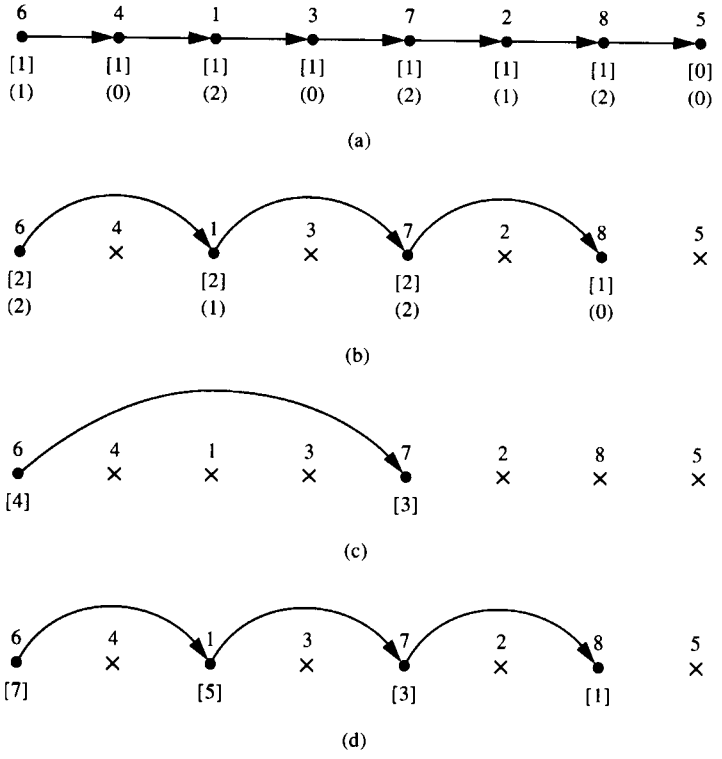


FIGURE 3.2
A linked list and its processing by Algorithm 3.3. (a) The initial list; the initial R values are given in brackets, and a 3-coloring is shown in parentheses. (b), (c) The lists obtained during the contraction process. (d) Restoration of the nodes removed from the list in (c).

To estimate the resources, we begin by bounding the number of iterations performed at step 2. Since the size of the independent set I is $\geq \frac{n_k}{5}$, where n_k is the size of the list at the beginning of the $(k+1)$st iteration of the **while** loop, we have $n_{k+1} \leq \frac{4}{5}n_k$, and hence $n_k \leq \left(\frac{4}{5}\right)^k n$. Therefore, the number of iterations required to reduce the size of the list below $n/\log n$ is $O(\log \log n)$. We now estimate the resources required to implement each iteration of step 2.

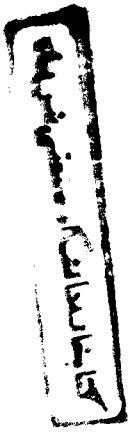
The optimal 3-coloring algorithm runs in $O(\log n)$ time using a linear number of operations (Theorem 2.7); hence, step 2.2 can be performed within these bounds. We have already analyzed the resources required for step 2.3 (Lemma 3.2). Step 2.4 requires labeling each of the nodes by 1 or 0

and performing a prefix-sums computation; hence, it takes $O(\log n)$ time, using a linear number of operations. Therefore, the $(k + 1)$st iteration of the **while** loop can be performed in $O(\log n)$ time, using $O(n_k)$ operations.

We conclude that the total running time of step 2 is $O(\log n \log\log n)$. The total number of operations performed by step 2 is $O(\sum_k n_k) = O\left(\sum_k \left(\frac{4}{5}\right)^k n\right) = O(n)$.

We have already seen that step 3 can be executed in $O(\log n)$ time, using $O(n)$ operations. Step 4 is straightforward once we have stored the appropriate information concerning the removed nodes in step 1. Therefore the resulting algorithm is optimal and runs in $O(\log n \log\log n)$ time. □

PRAM Model: The main procedures used in Algorithm 3.3 (coloring, prefix sums, Algorithm 3.1) do not require simultaneous memory access. It is easy to check that the remaining operations can also be performed without concurrent memory access. Therefore, Algorithm 3.3 runs on the EREW PRAM model. □



Remark 3.3: The list-ranking algorithm (Algorithm 3.3) can be applied to a given list or to that list's reverse (defined by the predecessor array); hence, the elements can be ranked efficiently from either end of the list. As mentioned in Section 2.2, the **parallel prefix** problem is similar to the prefix-sums problem, except that the elements a_i appear in the nodes of a linked list. That is, each node of the linked list contains a data element and a pointer to its successor, and we want to compute all partial sums s_k, where $1 \le k \le n$ and s_k is the sum of elements held in the nodes of ranks between 1 and k (the *rank* here refers to the distance from the beginning of the list).

The parallel prefix problem can be solved by the list-ranking algorithm if we initialize $R(i) = a_i$ and use the predecessor relation. Another method would be to rank the elements of the list by using the list-ranking algorithm, to store the elements in consecutive memory locations in their ranked order, and then to apply the prefix-sums algorithm. In either case, the parallel prefix can be performed within the same complexity bounds as those of the list-ranking problem. □

It turns out that a new technique is required to perform the initial list contraction optimally in $O(\log n)$ time. We devote the next section to this technique. The reader may wish to skip Section 3.1.2, but should note that an optimal $O(\log n)$ time list ranking algorithm will be assumed for the remainder of the chapter, including the exercises.

3.1.2 *AN OPTIMAL O(LOG n) TIME LIST-RANKING ALGORITHM

The weakness of simple optimal list-ranking algorithm (Algorithm 3.3) lies in performing the costly operations of identifying a large independent set and of

compacting the list during each iteration of the list-contraction process (step 2). We develop an alternative method to shrink the list of n nodes to $O(n/\log n)$ nodes without having to perform either operation. The key ideas behind this method are given next. For clarity, we assume that $\log n$ is an integer that divides n.

Overall Strategy. We divide the array S into $n/\log n$ subarrays $\{B_i\}$, called **blocks,** each containing $\log n$ items. Each block B_i will be processed sequentially but concurrently with all the other blocks. An index $p(i)$ will be used to point to a node in B_i. We denote this node by $N(p(i))$. Initially, $p(i) = (i - 1)\log n + 1$, for $1 \leq i \leq n/\log n$; that is, $p(i)$ points to the first node in each block B_i. As we proceed with our algorithm, however, the blocks will be processed at different speeds; hence, we need the pointers $p(i)$.

We consider the initial nodes $N(p(i))$ and label each $N(p(i))$ as **active,** for $1 \leq i \leq n/\log n$; we label each of the remaining nodes as **inactive.** If the active nodes form an independent set (that is, neither the successor node nor the predecessor node of each $N(p(i))$ is active), then these nodes can be removed from the list immediately (recall Algorithm 3.2 and Remark 3.2). We can then advance each pointer $p(i)$ to the next consecutive location of block B_i, that is, we set $p(i) = p(i) + 1$. In general, however, only a subset of the active nodes, called the **isolated** nodes, will form an independent set, whereas the rest of the active nodes will form sublists of the original list.

EXAMPLE 3.3:

A 16-node list and some of the pointers are shown in Fig. 3.3. In this case, we have four blocks B_1, B_2, B_3, and B_4. Initially, the active nodes are 1, 5, 9, and 13; these are indicated by the four pointers, $N(p(1)) = 1, N(p(2)) = 5$, and so on. The link information of the active nodes is indicated by the arcs. For example, $S(1) = 6, S(5) = 13, S(9) = 5, S(13) = 16$. Hence, node 1 is the only isolated node, since its predecessor node and its successor node are not active; the remaining active nodes form the sublist $9 \rightarrow 5 \rightarrow 13$. □

The contraction procedure consists of $O(\log n)$ stages, in which each stage can be implemented in $O(1)$ time, using $O(n/\log n)$ operations. Initially, there is a pointer $p(i)$ to the first node in B_i such that $N(p(i))$ is labeled active. All other nodes are labeled inactive. We now describe the first stage of our procedure. All the remaining stages are identical; a description of one will be given later.

First Stage. Each isolated active node $N(p(i))$ is removed from the list. Each remaining active node is then part of a certain sublist. Note that such a sublist may be of size $\Omega(n/\log n)$. We proceed to show how to break each

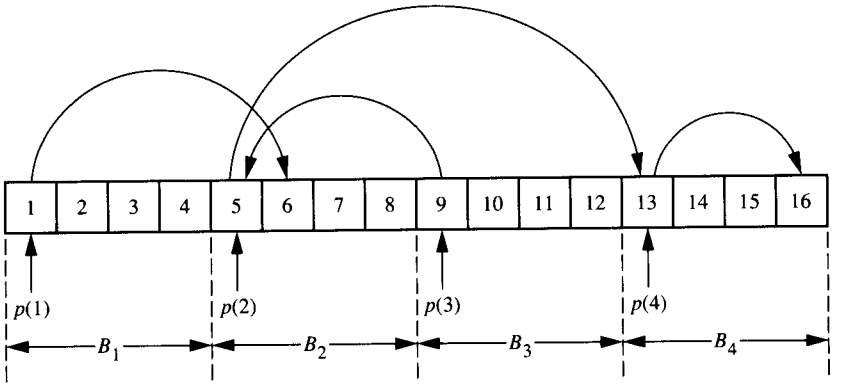


FIGURE 3.3
A list as partitioned by the contraction algorithm. In this case, there are four blocks such that initially $p(i)$ points to the first element in B_i. The arcs indicate some of the link information of the given list.

sublist into short *chains* in $O(1)$ time using a linear number of operations (in the size of the remaining active nodes).

We apply the basic coloring procedure (Algorithm 2.9) given in Section 2.7 twice to each remaining active node, starting with the initial coloring $c(i) = i$. Recall that this coloring procedure consists of determining the least significant bit position k in which $c(i)$ and $c(S(i))$ disagree, and setting the new color of i to be $c'(i) = 2k + c(i)_k$. Therefore, the remaining active nodes are colored properly with $O(\log \log n)$ colors. Each node whose color is a local minimum is labeled as a **ruler,** and each of the remaining active nodes is labeled as a **subject.** Figure 3.4 illustrates what happens to each active node during the first stage.

Thus, each of the nodes of a sublist is labeled either a ruler or a subject. The rulers of a sublist partition it into **chains**; each chain consists of a ruler r followed by the subjects between r and the next ruler (or the end of the sublist, if the next ruler does not exist). Without loss of generality, we assume that the leftmost node on a sublist (that is, an active node whose predecessor is not active) is a ruler, because otherwise we make it into a ruler. The size of each such chain is $O(\log \log n)$.

We end the first stage by advancing the pointer of each subject and each removed node, and labeling the corresponding new nodes as active.

General Stage. The input to a general stage consists of at most $n/\log n$ pointers $p(i)$, each $p(i)$ pointing to an element in block B_i. Each node $N(p(i))$

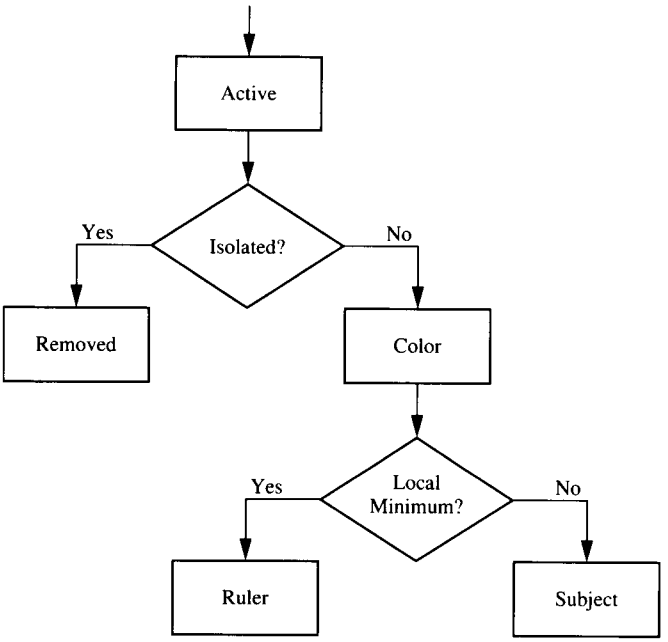


FIGURE 3.4
The state of each active node during the first stage of the list-contraction algorithm. Each such node either is removed, or becomes a ruler or a subject.

is labeled either active or a ruler. If $N(p(i))$ is a ruler, then the next subject (given by the original successor relation) is removed and is labeled removed. The node $N(p(i))$ becomes an active node if the removed node was its last subject. We now proceed just as in the first stage by removing active nodes that are isolated and breaking each sublist of the remaining active nodes into chains of length $O(\log \log n)$. Figure 3.5 illustrates what happens to a ruler or an active node during a general stage.

Finally, the pointers of the removed and the subject nodes are advanced, and the corresponding new nodes are labeled active.

List-Contraction Algorithm. The algorithm describing a general stage performed by processor P_i is given next, where $1 \leq i \leq n/\log n$. We do not give our algorithm in the WT presentation level because it is just as easy to describe it in terms of the sequence of operations to be performed by each processor. Processor P_i is responsible for processing block B_i.

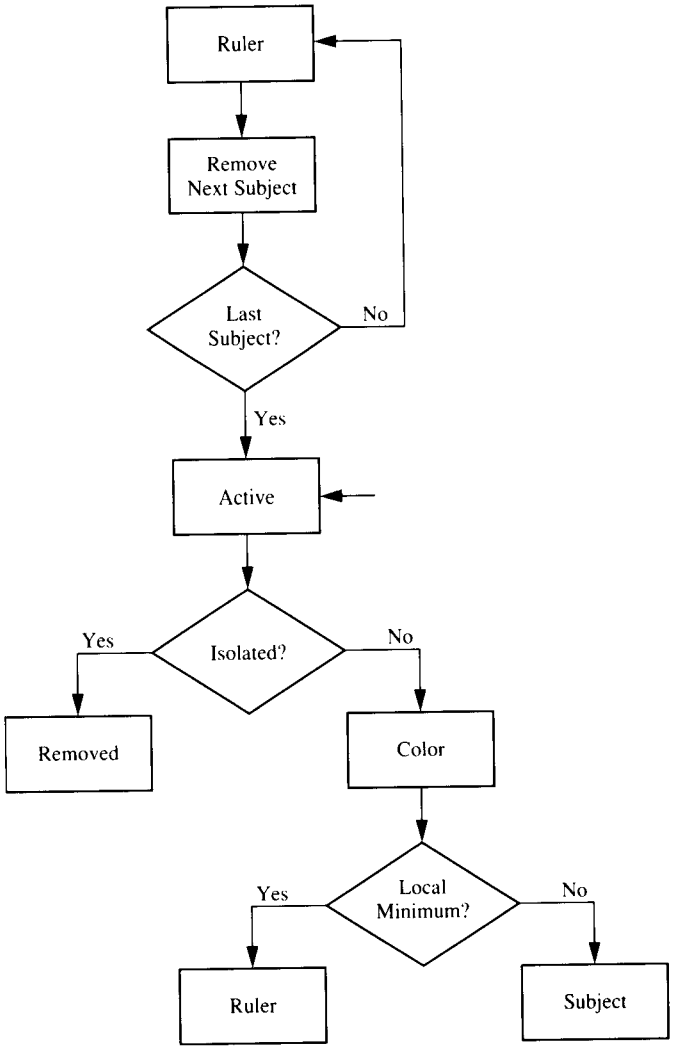


FIGURE 3.0
The state of each active or ruler node during a general stage of the list-contraction algorithm. A ruler can either stay a ruler or become an active node. An active node either is removed, or becomes a ruler or a subject.

ALGORITHM 3.4
(Contracting a List)

Input: *A linked list with n nodes given by the successor array S, which is partitioned into n/log n blocks $\{B_i\}$ such that (1) a pointer $p(i)$ to a node $N(p(i))$ in block B_i is given; (2) each node $N(p(i))$ is labeled either active or a ruler; and (3) each of the remaining nodes is labeled either inactive or a subject.*

Output: *A contracted list such that (1) a subject from each chain and all isolated active nodes are removed; (2) each nonempty block B_i has a pointer $p(i)$ such that $N(p(i))$ is labeled either active or a ruler.*

A general stage: (Algorithm for Processor P_i)

begin

1. *If $N(p(i))$ is a ruler, remove the next subject. Label $N(p(i))$ active if this was the last subject in its chain.*
2. *If $N(p(i))$ is active and isolated, remove $N(p(i))$ and label $N(p(i))$ as removed.*
3. *If $N(p(i))$ is active, then color $N(p(i))$ by applying the basic coloring procedure (Algorithm 2.9) twice. Label $N(p(i))$ as a ruler if its color is a local minimum. Otherwise, label $N(p(i))$ as a subject.*
4. *If $N(p(i))$ is labeled removed or a subject, then set $p(i) = p(i) + 1$. If $p(i)$ crosses the border of B_i, then exit. Otherwise, label $N(p(i))$ as active.*

end

Lemma 3.4: *Algorithm 3.4 shrinks the input list by removing the isolated active nodes and a subject from each chain, and creates new active nodes and possibly new chains of length $O(\log \log n)$ such that each nonempty block B_i has a pointer $p(i)$ with $N(p(i))$ labeled either active or a ruler. The algorithm can be implemented in $O(1)$ time, using $O(n/\log n)$ operations.* □

Complexity Bounds of the List-Contraction Algorithm. It remains to be shown that, after $O(\log n)$ iterations, the number of remaining nodes is $O(n/\log n)$. To demonstrate this fact, we need to discuss an additional technical detail related to the process used in selecting the rulers and the subjects at each stage. This detail is needed to carry out the analysis on the number of iterations.

Suppose we have a chain of length $O(\log \log n)$ formed by the nodes $N(p(i_1)), N(p(i_2)), \cdots, N(p(i_k))$. Let $\alpha(i_j)$ be the index of $N(p(i_j))$ within its block (starting from 0 for the first element up to $\log n - 1$ for the last element of the block). We would like the ruler of a chain to be the node with the largest index, say $\alpha(i_l) = \max_{1 \le j \le k}\{\alpha(i_j)\}$. This has to be accomplished in $O(1)$ time using a linear number of operations. We can take each *local maximum* with respect to the indices $\alpha(i_j)$'s to be a ruler. The subjects associated with such

a ruler are located between two local minima that trap the ruler (one of the local minima may not exist). Figure 3.6 illustrates the possible structure of a chain with respect to the indices $\alpha(i_j)$'s.

Since we can traverse the list backward (by using the predecessor array P) instead of forward to remove the subjects, each stage can still be executed in $O(1)$ time, using a linear number of operations. In the analysis given next, we assume that the index of the ruler of each chain is the maximum among the indices of its subjects, and that each chain is of length less than or equal to $\log \log n$.

We are now ready to show that after $O(\log n)$ stages, the number of the remaining nodes is $O(n/\log n)$.

Lemma 3.5: *After $O(\log n)$ stages of Algorithm 3.4, the number of remaining nodes is $O(n/\log n)$.*

Proof: Since the nodes of the different blocks are processed at different rates, the analysis depends on an accounting scheme that assigns a weight to each node depending on that node's location within its block. Let $q = 1/\log \log n$. Initially, each block has $\log n$ nodes. The ith node from the top of its block is assigned a weight of $(1 - q)^i$, for $0 \leq i < \log n$. Therefore, as we move down a block, the weight of a node decreases and thus its contribution becomes less important. The total weight of each block is $\sum_{0 \leq i < \log n} (1 - q)^i < 1/q$, and the total weight of all the items is less than $n/q \log n$.

We will show that, after $O(\log n)$ stages, the weight of the elements remaining in the list is at most $(n/\log n)\ (1 - q)^{\log n}$, which implies that $O(n/\log n)$ elements remain, since the smallest weight is $(1 - q)^{\log n - 1}$.

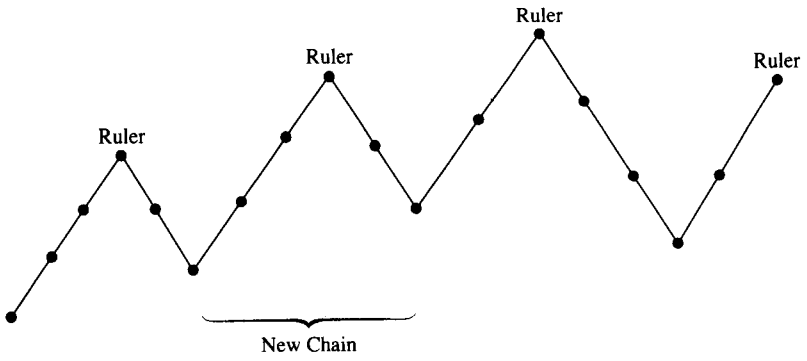


FIGURE 3.6
The structure of the sequence $\alpha(i_j)$ corresponding to a chain. An example of a new chain and its ruler is illustrated.

Claim: After each stage, the total weight of the elements that have not been removed is reduced by at least a factor $1 - (q/4)$.

Proof of Claim: We distribute the total weight of the elements remaining in the list among the blocks as follows. With each block B_i, we associate not only the elements remaining in B_i, but also the subjects of $N(p(i))$, if $N(p(i))$ happens to be a ruler. The *weight of a block* is the sum of the weights of all the elements associated with that block. Figure 3.7(a) illustrates the elements associated with B_i when $N(p(i))$ is an active node; Figure 3.7(b) covers the case where $N(p(i))$ is a ruler. Notice that, during any stage, each remaining node is associated with exactly one block.

Three separate cases can occur:

1. *An active node is removed as an isolated node* (step 2 of Algorithm 3.4). Suppose this node is the ith node. The weight of its block has been reduced from $\sum_{i \le j < \log n}(1-q)^j$ to $\sum_{i+1 \le j < \log n}(1-q)^j$, which is less than $(1-q)\sum_{i \le j < \log n}(1-q)^j \le (1-\frac{q}{4})\sum_{i \le j < \log n}(1-q)^j$.
2. *An active node is labeled as a subject* (step 3 of Algorithm 3.4). Consider the chain containing this node and all the associated blocks. The weight of a node will be reduced by a factor of ½ when the node becomes a

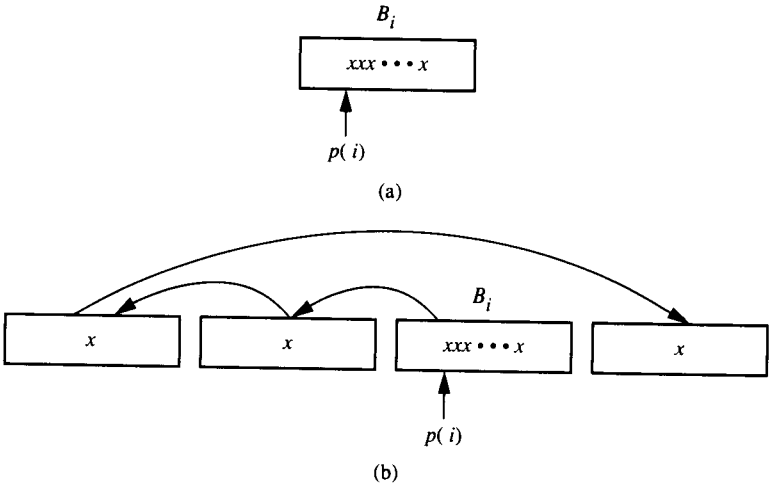


FIGURE 3.7
The elements marked by x are associated with block B_i when (a) $N(p(i))$ is active and (b) when $N(p(i))$ is a ruler.

subject, and the remaining ½ will be accounted for when the subject is removed. Suppose the weight of the ruler is $(1 - q)^{i_1}$, and the weights of the subjects in the chain are $(1 - q)^{i_j}$, where $2 \le j \le k$ for some k. As we have seen, we can assume i_1 to be the largest index among the i_j's. The total weight of all the blocks associated with this chain is $Q = \sum_{j=1}^{k} \sum_{i_j \le l < \log n} (1 - q)^l$. The selection of the subjects reduces the weight to $Q - \frac{1}{2}\sum_{j=2}^{k} (1 - q)^{i_j}$. Notice that $\sum_{i_j \le l < \log n} (1 - q)^l < \frac{1}{q} (1 - q)^{i_j}$ (the left-hand side is a geometric series). Hence, $Q < \frac{1}{q}\sum_{j=1}^{k} (1 - q)^{i_j} \le \frac{2}{q}\sum_{j=2}^{k} (1 - q)^{i_j}$, since i_1 is the largest index. This latter inequality implies that $qQ/4 \le (1/2)\sum_{j=2}^{k}(1 - q)^{i_j}$, and therefore $Q - \frac{1}{2}\sum_{j=2}^{k} (1 - q)^{i_j} \le (1 - \frac{q}{4})Q$.

3. *The current node is a ruler and removes a subject* (step 1 of Algorithm 3.4). Let the ruler be of weight $(1 - q)^{i_1}$, and the subjects be of weights $(1 - q)^{i_2}, \ldots, (1 - q)^{i_k}$. Without loss of generality, assume the element of heaviest weight—say, of weight $(1 - q)^{i_2}$—is removed. If the element removed is not the heaviest, we can rearrange the weights so as to assign the heaviest weight to the removed node. The total weight of the block and the subjects of the ruler is given by $Q = \sum_{i_1 \le l < \log n} (1 - q)^l + \frac{1}{2}\sum_{2 \le j \le k} (1 - q)^{i_j}$. After we have removed the subject, the weight reduces to $Q - \frac{1}{2} (1 - q)^{i_2}$. As before, $\sum_{i_1 \le l < \log n} (1 - q)^l < (1/q)(1 - q)^{i_1}$; hence, $\sum_{i_1 \le < \log n}(1 - q)^l < (1/q)(1 - q)^{i_2}$. Moreover, $(1/2)\sum_{2 \le j \le k} (1 - q)^{i_j} \le \frac{k}{2} (1 - q)^{i_2}$ since $(1 - q)^{i_2}$ is the heaviest element among the subjects. Since the length k of each chain is less than or equal to $\log \log n = 1/q$, we obtain $\frac{1}{2}\sum_{2 \le j \le k}(1 - q)^{i_j} \le (1/2q)\ (1 - q)^{i_2}$. It follows that $Q \le (3/2q)(1 - q)^{i_2}$, which in turn implies that $Q - \frac{1}{2}(1 - q)^{i_2} \le \left(1 - \frac{q}{3}\right)Q$.



Since, at each stage, the total weight is reduced by a factor of at least $(1 - (q/4))$, we see that after 5 log n stages, the total weight is reduced to $(n/q \log n)(1 - (q/4))^{5 \log n}$. It can be shown that $(n/q \log n)(1 - (q/4))^{5 \log n} < (n/\log n)(1 - q)^{\log n}$, for large enough n (see Exercise 3.6). Thus, the proof follows, since the smallest weight is $(1 - q)^{\log n - 1}$. □

Theorem 3.2: *The list-ranking problem on a list with n nodes can be solved in* $O(\log n)$ *time, using a linear number of operations.*

Proof: Each stage of the contraction algorithm (Algorithm 3.4) requires $O(1)$ time using $O(n/\log n)$ operations. After $O(\log n)$ iterations, the number of nodes remaining in the list is $O(n/\log n)$. Thus, the whole contraction process takes $O(\log n)$ time, using $O(n)$ operations. We can now apply the pointer

jumping technique to rank all the remaining elements in $O(\log n)$ time, using $O(n)$ operations. As we have seen before, the reverse process can be completed within these bounds. □

PRAM Model: Algorithm 3.4 does not require any concurrent memory access. Hence, the optimal $O(\log n)$ time list-ranking algorithm runs on the EREW PRAM model. □

Next, we introduce the Euler-tour technique that uses the optimal $O(\log n)$ time list-ranking algorithm.

3.2 The Euler-Tour Technique

Let $T = (V, E)$ be a given tree and let $T' = (V, E')$ be the directed graph obtained from T when each edge $(u, v) \in E$ is replaced by two arcs $\langle u, v\rangle$ and $\langle v, u\rangle$. Since the indegree of each vertex of T' is equal to its outdegree, T' is an **Eulerian** graph; that is, it has a directed circuit that traverses each arc exactly once. It turns out that an **Euler circuit** of T' can be used for the optimal parallel computation of many functions on T. We start by examining how we obtain such a circuit; we then use it to develop optimal parallel algorithms for computing several tree functions.

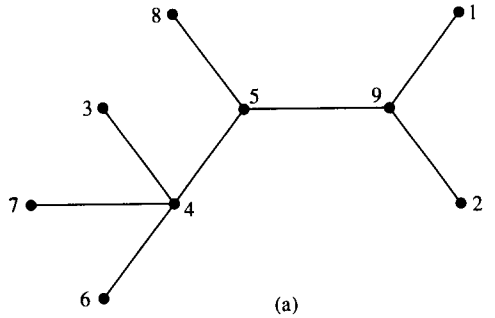
3.2.1 EULER CIRCUITS

An Euler circuit of $T' = (V, E')$ can be defined by specifying the **successor function** s mapping each arc $e \in E'$ into the arc $s(e) \in E'$ that follows e on the circuit. There is a simple way to introduce a suitable successor function. For each vertex $v \in V$ of the tree $T = (V, E)$, we fix a certain ordering on the set of vertices adjacent to v—say, $adj(v) = \langle u_0, u_1, \ldots, u_{d-1}\rangle$, where d is the degree of v. We define the successor of each arc $e = \langle u_i, v\rangle$ as follows: $s(\langle u_i, v\rangle) = \langle v, u_{(i+1) \bmod d}\rangle$, for $0 \le i \le d - 1$.

EXAMPLE 3.4:

A tree $T = (V, E)$ and an ordering of the vertices adjacent to each vertex v are shown in Fig. 3.8(a) and (b). The successor function gives rise to the table shown in Fig. 3.8(c). Consider, for instance, the vertex 5 whose adjacent vertices are given by $adj(5) = \langle 8, 4, 9\rangle$. Such a list implies that $s(\langle 8, 5\rangle) = \langle 5, 4\rangle$, $s(\langle 4, 5\rangle) = \langle 5, 9\rangle$, and $s(\langle 9, 5\rangle) = \langle 5, 8\rangle$ as indicated in the successor table. The

reader should verify that the successor table does indeed specify an Euler circuit of the directed graph $T' = (V, E')$. Starting with the arc $\langle 9, 1\rangle$, the circuit defined by the successor function is shown in Fig. 3.8(d). □



(a)

v	$adj(v)$
1	9
2	9
3	4
4	5, 3, 7, 6
5	8, 4, 9
6	4
7	4
8	5
9	5, 2, 1

(b)

Arc	Successor
⟨9, 1⟩	⟨1, 9⟩
⟨9, 2⟩	⟨2, 9⟩
⟨4, 3⟩	⟨3, 4⟩
⟨5, 4⟩	⟨4, 3⟩
⟨3, 4⟩	⟨4, 7⟩
⟨7, 4⟩	⟨4, 6⟩
⟨6, 4⟩	⟨4, 5⟩
⟨8, 5⟩	⟨5, 4⟩
⟨4, 5⟩	⟨5, 9⟩
⟨9, 5⟩	⟨5, 8⟩
⟨4, 6⟩	⟨6, 4⟩
⟨4, 7⟩	⟨7, 4⟩
⟨5, 8⟩	⟨8, 5⟩
⟨5, 9⟩	⟨9, 2⟩
⟨2, 9⟩	⟨9, 1⟩
⟨1, 9⟩	⟨9, 5⟩

(c)

⟨9, 1⟩ → ⟨1, 9⟩ → ⟨9, 5⟩ → ⟨5, 8⟩ → ⟨8, 5⟩ → ⟨5, 4⟩ →
⟨4, 3⟩ → ⟨3, 4⟩ → ⟨4, 7⟩ → ⟨7, 4⟩ → ⟨4, 6⟩ → ⟨6, 4⟩ →
⟨4, 5⟩ → ⟨5, 9⟩ → ⟨9, 2⟩ → ⟨2, 9⟩ → ⟨9, 1⟩

(d)

FIGURE 3.8
The Euler circuit of the tree for Example 3.4. (a) An input tree $T = (V, E)$; (b) an ordering of the vertices adjacent to each vertex; (c) the successor function defined on the set of arcs; (d) the resulting Euler circuit, starting with the arc $\langle 9, 1\rangle$.

Lemma 3.6: *Given a tree $T = (V, E)$ and an ordering of the set of vertices adjacent to each vertex $v \in V$, the function s defined previously specifies an Euler circuit in the directed graph $T' = (V, E')$, where E' is obtained by replacement of each $e \in E$ by two arcs of opposite directions.*

Proof: We have already observed that the directed graph $T' = (V, E')$ is Eulerian. Given the successor function s, each arc $e \in E'$ is assigned a unique successor and is the successor of a unique arc. It remains to be shown that s defines one cycle and not a set of arc-disjoint cycles. The proof is by induction on the number n of vertices.

The base case $n = 2$ is straightforward. Hence, let $n > 2$, and assume that the induction hypothesis holds for any tree with less than n vertices.

Let $T = (V, E)$ be a tree with n vertices, and let $v \in V$ be a leaf. Denote by u the vertex adjacent to v. It follows that $adj(v) = \langle u \rangle$, which implies that $s(\langle u, v \rangle) = \langle v, u \rangle$. Suppose $adj(u) = \langle \ldots, v', v, v'', \ldots \rangle$, where v' and v'' could be the same (see Fig. 3.9). In this case, we have $s(\langle v', u \rangle) = \langle u, v \rangle$ and $s(\langle v, u \rangle) = \langle u, v'' \rangle$.

Remove v from V. Hence, $adj(u)$ becomes the list $\langle \ldots, v', v'', \ldots \rangle$. For the resulting new tree T_1, the successor of each of its arcs remains the same as before, except for the arc $\langle v', u \rangle$, whose new successor is given by $\langle u, v'' \rangle$. By the induction hypothesis, the successor function s defines an Euler circuit on T_1. Replacing $s(\langle v', u \rangle) = \langle u, v'' \rangle$ by $s(\langle v', u \rangle) = \langle u, v \rangle$ and $s(\langle v, u \rangle) = \langle u, v'' \rangle$ results in an Euler circuit for T', and therefore the lemma follows. □

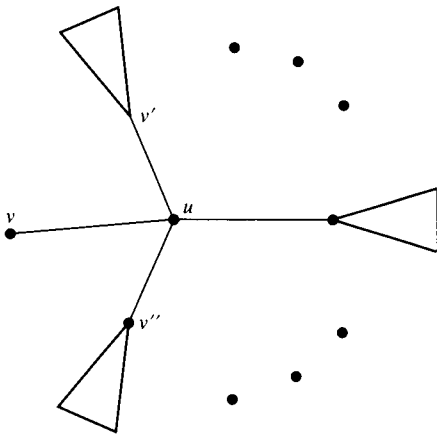


FIGURE 3.9
A leaf v in a tree T and the vertices adjacent to the vertex u.

In summary, given a tree $T = (V, E)$ and an ordering of the set of vertices adjacent to each vertex v, an Euler circuit of the directed graph $T' = (V, E')$ can be easily specified. We call the Euler circuit of T' the **Euler tour** of T.

We now turn into the algorithmic aspects of constructing an Euler tour from the given tree $T = (V, E)$. Notice that the degree of each vertex $v \in V$ is not necessarily bounded, and no assumption is made concerning whether or not the tree is rooted.

A possible representation of T consists of the **adjacency lists** of the vertices. More precisely, for each vertex $v \in V$, the vertices adjacent to v are given in a linked list—say, $L[v] = \langle u_0, u_1, \cdots, u_{d-1} \rangle$—in some order, where d is the degree of v. In the directed graph $T' = (V, E')$, $L[v]$ can be viewed as representing the outgoing arcs of v—that is, the arcs $\langle v, u_0 \rangle$, $\langle v, u_1 \rangle$, $\cdots$, $\langle v, u_{d-1} \rangle$. In other words, the node of the list $L[v]$ containing u_i uniquely represents the arc $\langle v, u_i \rangle$. Since v appears on the lists $L[u_i]$, the corresponding nodes represent the arcs $\langle u_i, v \rangle$ for $0 \leq i \leq d - 1$. Therefore the adjacency lists $L[v]$'s represent uniquely all the arcs in the directed graph $T' = (V, E')$. In addition, the linked list $L[v]$ implies an ordering on the set of vertices adjacent to v, for each $v \in V$. Figure 3.10 shows a tree and the corresponding linked lists.

Consider the successor function s introduced previously, using the ordering implied by the adjacency lists. Then, $s(\langle u_i, v \rangle) = \langle v, u_{(i+1) \bmod d} \rangle$, where u_i is the ith vertex on $L[v]$, where $0 \leq i \leq d - 1$.

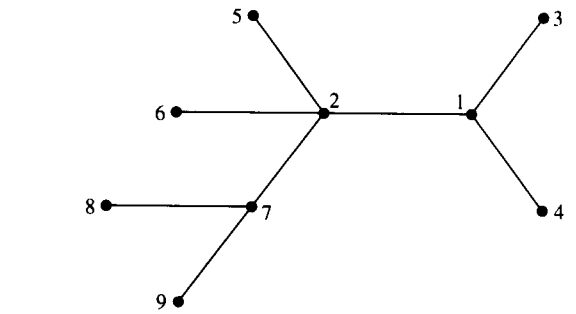
Given the node containing vertex u_i, we need two pieces of information to determine the successor function: (1) the arc $\langle u_i, v \rangle$, and (2) the successor $u_{(i+1) \bmod d}$ of u_i on the list $L[v]$ (which uniquely identifies the arc $\langle v, u_{(i+1) \bmod d} \rangle$). The successor information is immediately available except when u_i is at the end of the list. We can handle the latter case by making each adjacency list *circular*. To obtain arc $\langle u_i, v \rangle$, we assume that the node holding vertex u_i in the list $L[v]$ has an additional pointer to the node containing v in $L[u_i]$ (which uniquely represents the arc $\langle u_i, v \rangle$).

It follows that a node on each circular adjacency list consists of a vertex, say u, and two **pointers.** One pointer can be used to deduce the vertex u' following u on the adjacency list of some vertex v, and hence arc $\langle v, u' \rangle$. The other pointer can be used to obtain the arc $\langle u, v \rangle$ of T'.

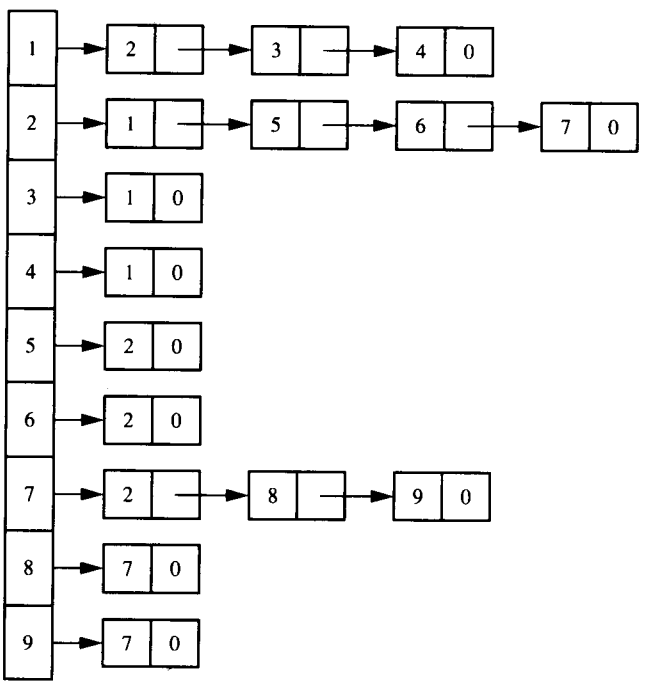
EXAMPLE 3.5:

A tree T and its adjacency lists with the additional pointers are shown in Fig. 3.11. Consider an arbitrary vertex—say, vertex 5. Vertex 5 appears on the list $L[2]$. The node containing vertex 5 has a pointer to a node containing vertex 2. Hence, we can use this pointer to deduce the arc $\langle 5, 2 \rangle$ of T'. We can also use this node to conclude that vertex 6 comes after vertex 5 in the adjacency list of vertex 2, and to deduce arc $\langle 2, 6 \rangle$. □

We are now ready for the following theorem.



(a)



(b)

FIGURE 3.10
A tree and the corresponding linked list. (a) A tree T. (b) The adjacency lists of T, where each vertex v has a pointer to a linked list containing the vertices adjacent to v.

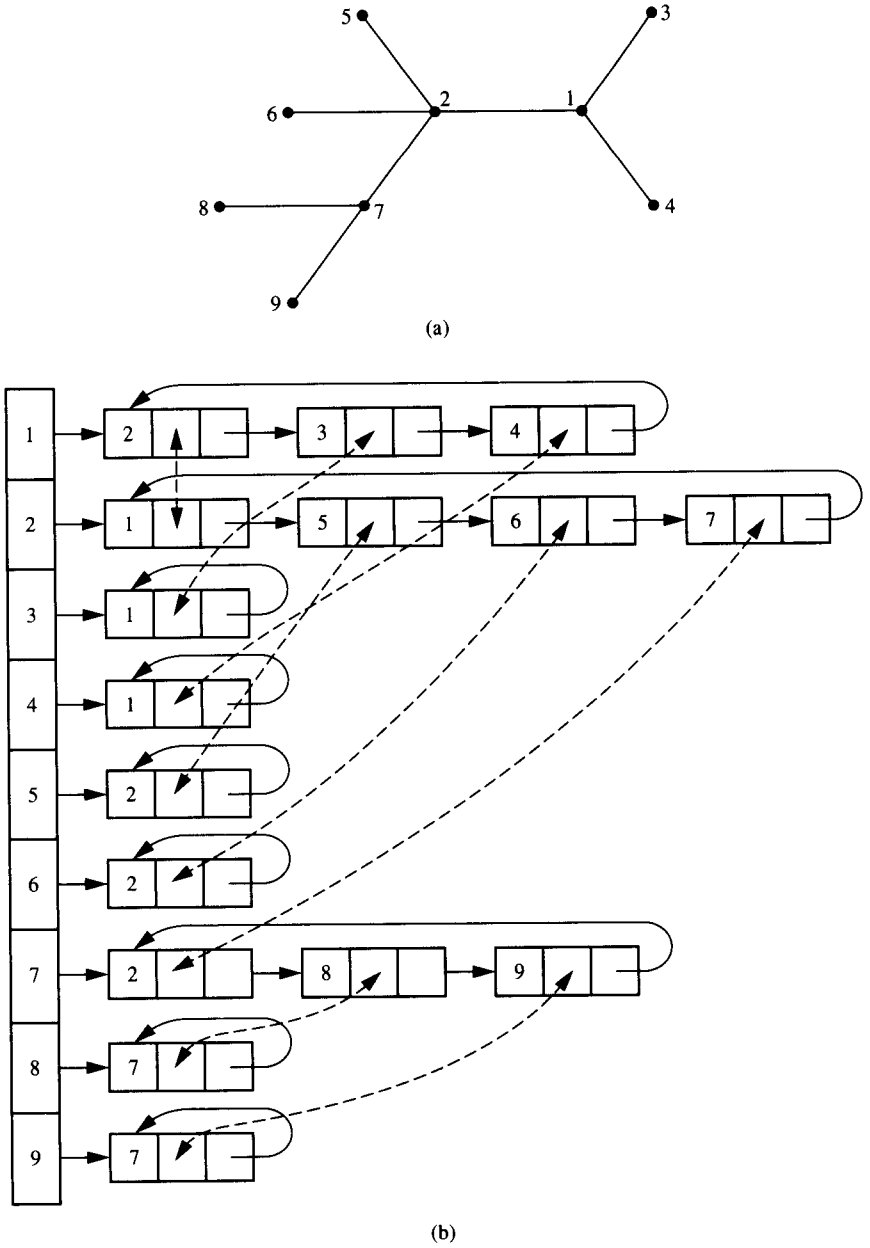


FIGURE 3.11
The tree and the adjacency lists for Example 3.5. (a) An input tree; (b) The tree's representation by the circular adjacency lists with the additional pointers. Each node consists of a vertex and two pointers.

Theorem 3.3: *Given a tree $T = (V, E)$ defined by the adjacency lists of its vertices with the additional pointers as described previously, we can construct an Euler circuit in T' in $O(1)$ time using $O(n)$ operations, where $|V| = n$.*

Proof: Consider an arbitrary node of the adjacency lists. This node holds a vertex—say u—and a pointer to its successor holding, say vertex u', and another pointer to a node holding a vertex v. Clearly, $\langle u, v\rangle$ is an arc such that $s(\langle u, v\rangle) = \langle v, u'\rangle$. For each node, this operation can be carried out in $O(1)$ time, since the pointers in the node holding vertex u uniquely identify arcs $\langle u, v\rangle$ and $\langle v, u'\rangle$. Therefore, the Euler circuit can be generated in $O(1)$ time, using a linear number of operations for the given input representation. □

PRAM Model: We can obtain the successor of each arc by examining each node separately. Hence, the procedure to compute the Euler tour does not require any simultaneous memory access. It follows that computing the Euler tour can be performed on the EREW PRAM model within the stated bounds. □

Remark 3.4: *From now on, whenever the Euler tour is used, the input tree is assumed to be represented by the set of circular adjacency lists with the additional pointers as we have described.* □

3.2.2 TREE COMPUTATIONS

We now discuss several applications of the Euler-tour technique to computing tree functions.

Rooting a Tree. The Euler tour of a tree $T = (V, E)$ can be used to process T and to compute several functions on T efficiently. We start with the problem of **rooting** T at a vertex r; that is, for each vertex $v \neq r$, we determine the parent $p(v)$ of v when T is rooted at r.

As before, let $T' = (V, E')$ denote the directed graph obtained from T by replacing each edge of E by two arcs of opposite directions. We find the Euler tour of T by computing the successor function s. Let the adjacency list of vertex r be given by $L[r] = \langle u_0, u_1, \ldots, u_{d-1}\rangle$. Break the Euler tour at r by setting $s(\langle u_{d-1}, r\rangle) = 0$. We now have a directed Euler path in T' that begins at r, visits each arc exactly once, and ends at r.

Consider the ordered list EP of the arcs on the Euler path $EP = (e_1 = \langle r, u_0\rangle, e_2 = s(e_1), s(e_2), \ldots, \langle u_{d-1}, r\rangle)$. The list EP can be viewed as the process of traversing the tree T as follows. We begin by visiting the root r, followed by visiting the vertex u_0 adjacent to r. The same search process is applied to the subtree rooted at u_0. Once all the vertices in the subtree rooted at u_0 are explored, we visit u_1, and proceed as before. The reader should

verify that the traversal induced by the Euler path EP is a **depth-first search** of the tree T starting at vertex r. In particular, confirm that arc $\langle p(u), u\rangle$ appears on the list EP *before* arc $\langle u, p(u)\rangle$, when the tree T is rooted at r.

EXAMPLE 3.6:

Consider again the tree $T = (V, E)$ shown in Fig. 3.11(a). Suppose we want to root T at vertex 1. We break the corresponding Euler circuit by setting $s(\langle 4, 1\rangle) = 0$. Starting from the root 1, the Euler path consists of a depth-first traversal of the vertices of the tree as follows: 1, 2, 5, 2, 6, 2, 7, 8, 7, 9, 7, 2, 1, 3, 1, 4, 1 (see Fig. 3.12). Notice, for example, that arc $\langle 2, 7\rangle$ appears before arc $\langle 7, 2\rangle$ on the Euler path, and that 2 is the parent of 7. □

The algorithm can now be stated more formally.

ALGORITHM 3.5
(Rooting a Tree)
Input: *(1) A tree T defined by the adjacency lists of its vertices, (2) an Euler tour defined by the successor function s, and (3) a special vertex r.*
Output: *For each vertex $v \neq r$, the parent $p(v)$ of v in the tree rooted at r.*
begin
1. Identify the last vertex u appearing on the adjacency list of r. Set $s(\langle u, r\rangle) = 0$.
2. Assign a weight of 1 to each arc $\langle x, y\rangle$, and apply parallel prefix on the list of arcs defined by s.
3. For each arc $\langle x, y\rangle$, set $x = p(y)$ whenever the prefix sum of $\langle x, y\rangle$ is smaller than the prefix sum of $\langle y, x\rangle$.
end

Lemma 3.7: *Given a tree $T = (V, E)$ represented by the circular adjacency lists with the additional pointers as described previously, and a special vertex r, we can root the tree T by specifying the parent $p(v)$ of each vertex $v \in V$ in $O(\log n)$ time, using a linear number of operations, where $|V| = n$.* □

Postorder Numbering. Another application of the Euler-tour technique is the processing of the vertices of a tree in a certain order. Suppose, for example, that we wish to determine the postorder traversal of the tree T rooted at r. The **postorder traversal** of T consists of the postorder traversals of the subtrees of r from left to right, followed by the root r. For our rooted tree $T = (V, E)$, we define the left-to-right ordering of the children of a vertex as the ordering implied by the Euler path EP. That is, the children of a vertex v are

visited (each for the first time) from left to right on the Euler path EP. Notice that the children of the root r appear in a left-to-right ordering in the adjacency list $L[r]$, but this ordering does not necessarily occur for the remaining vertices.

The goal is to determine the **postorder number** $post(v)$ of each vertex v—that is, v's rank in a postorder traversal of T.

The Euler path EP can be used to determine the postorder traversal of T as follows. The path EP visits each vertex v several times, the first time using the arc $\langle p(v), v\rangle$ and the last time using the arc $\langle v, p(v)\rangle$ after visiting all the descendants of v. Therefore, the ordered sublist of vertices obtained by retention of only the last occurrence of each vertex defines precisely the postorder traversal of the vertices of T. The postorder number of each vertex is computed by the following algorithm.

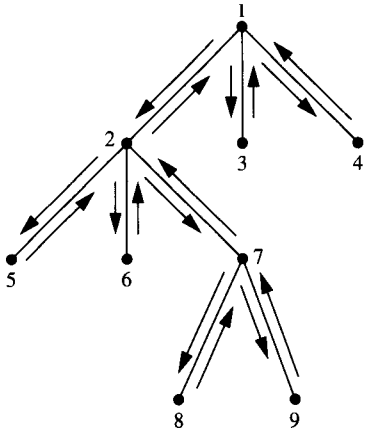
ALGORITHM 3.6
(Postorder Numbering)

Input: *(1) A rooted tree with root r, and (2) the corresponding Euler path defined by the function s.*
Output: *The postorder number post(v) of each vertex v.*
begin
1. For each vertex $v \neq r$, assign the weights $w(\langle v, p(v)\rangle) = 1$ and $w(\langle p(v), v\rangle) = 0$.
2. Perform parallel prefix on the list of arcs defined by s.
3. For each vertex $v \neq r$, set $post(v)$ equal to the prefix sum of $\langle v, p(v)\rangle$. For $v = r$, set $post(r) = n$, where n is the number of vertices in the given tree.
end

EXAMPLE 3.7:

Consider the tree of Fig. 3.12(a) (which is the same as the one shown in Fig. 3.11a) and its Euler tour. The weight and the prefix sum of each arc in the corresponding Euler path are shown in Fig. 3.12(b). From the prefix sums, we obtain the following: $post(5) = 1$, $post(6) = 2$, $post(8) = 3$, $post(9) = 4$, $post(7) = 5$, $post(2) = 6$, $post(3) = 7$, $post(4) = 8$. We also have $post(1) = 9$. The reader can verify that this numbering corresponds to a postorder traversal of the given tree. □

Computing the Vertex Level. Another useful operation is to compute the **level** $level(v)$ of each vertex v, which is the distance (number of edges) between v and the root r. We again use the Euler path of T rooted at r. We assign $w(\langle p(v), v\rangle) = +1$, and $w(\langle v, p(v)\rangle) = -1$, and perform parallel prefix on the list defining the Euler path. Then, we set $level(v)$ to be equal to the prefix sum of $\langle p(v), v\rangle$.



(a)

Euler Path	Weight	Prefix Sums
⟨1, 2⟩	0	0
⟨2, 5⟩	0	0
⟨5, 2⟩	1	1
⟨2, 6⟩	0	1
⟨6, 2⟩	1	2
⟨2, 7⟩	0	2
⟨7, 8⟩	0	2
⟨8, 7⟩	1	3
⟨7, 9⟩	0	3
⟨9, 7⟩	1	4
⟨7, 2⟩	1	5
⟨2, 1⟩	1	6
⟨1, 3⟩	0	6
⟨3, 1⟩	1	7
⟨1, 4⟩	0	7
⟨4, 1⟩	1	8

(b)

FIGURE 3.12
The tree and Euler tour for Example 3.7. (a) A tree rooted at vertex 1. (b) A table showing the arcs in the order they appear on the Euler path, the weight assigned to each arc, and the prefix sums needed for the postorder numbering or for computing the size function. The prefix sum of arc $\langle v, p(v)\rangle$ is the post-order number of vertex v, and $size(v)$ is equal to the difference between the prefix sums of $\langle v, p(v)\rangle$ and $\langle p(v), v\rangle$.

Computing the Number of Descendants. The two previous tree functions, along with two new ones, can be summarized in the following theorem.

Theorem 3.4: *Let T be a tree with n vertices rooted at r given by the adjacency lists as we have described. Each of the following computations can be done optimally in $O(\log n)$ time on the EREW PRAM.*

1. Computing the postorder number of each vertex
2. Computing the level of each vertex
3. Computing the preorder number of each vertex
4. Computing the number of descendants of each vertex

Proof: Claims 1 and 2 have already been established; claim 3 will be left to Exercise 3.13. We now prove claim 4 by showing how to compute the number of descendants of each vertex.

Let $size(v)$ be the number of vertices in the subtree rooted at v. Consider the Euler path EP defined by the successor function s. We apply the parallel prefix algorithm to the list EP using the weight $w(\langle p(v), v\rangle) = 0$ and $w(\langle v, p(v)\rangle) = 1$, for each vertex $v \neq r$. Then, $size(v)$ is equal to the difference between the prefix sums of $\langle v, p(v)\rangle$ and $\langle p(v), v\rangle$, since all the arcs appearing between $\langle p(v), v\rangle$ and $\langle v, p(v)\rangle$ on the Euler path EP belong to the subtree rooted at v ($v \neq r$). □

EXAMPLE 3.8:

We wish to determine the size function on the tree rooted at vertex 1 shown in Fig. 3.12. For each $v \neq 1$, arc $\langle v, p(v)\rangle$ is assigned a weight of 1, and each of the remaining arcs is assigned a weight of 0. The prefix sum of each node on the EP list is also shown. We can determine the size function by using these prefix sums. For example, $size(2)$ is equal to the difference between the prefix sum of $\langle 2, 1\rangle$ and the prefix sum of $\langle 1, 2\rangle$, which is equal to 6. Similarly, we can determine the size function for all the remaining vertices. □

3.3 Tree Contraction

In Section 3.2, we presented the Euler-tour technique and a few of its applications in computing tree functions. Other applications are covered in Section 3.4 and in Chapter 5. However, there are important tree problems that

cannot be solved efficiently with this technique alone. Consider, for example, the **expression evaluation problem.**

A given arithmetic expression is represented with a binary tree such that a constant is associated with each leaf, and an arithmetic operator—say, $+$, $-$, $\times$, or $\div$—is associated with each internal node. The goal is to compute the value of the expression at the root. On some occasions, we will also be interested in computing all the subexpressions defined at the internal nodes. This type of problem can be handled efficiently by **tree contraction.**

Tree contraction is a systematic way of shrinking a tree into a single vertex by successively applying the operation of merging a leaf with its parent or merging a degree-2 vertex with its parent. The expression evaluation problem provides a justification for such a process. A leaf u holds a constant value, which can be incorporated into the data stored at the parent $p(u)$. Removing a vertex v of degree 2 amounts to postponing the evaluation of the subexpression at v and incorporating its effect into the data stored at the parent $p(v)$. More details about the expression evaluation problem are provided in Section 3.3.1.

We next introduce an operation that captures this merging process.



3.3.1 THE *RAKE* OPERATION

Let $T = (V, E)$ be a rooted binary tree, with root r such that, for each vertex v, where $v \neq r$, $p(v)$ represents the parent of v. For clarity, we assume that each internal vertex has exactly two children.

We introduce the primitive operation, called **rake,** on T as follows. Given a leaf u such that $p(u) \neq r$, the *rake* operation applied at u consists of removing u and $p(u)$ from T and connecting the sibling of u, denoted by $sib(u)$, to $p(p(u))$.

EXAMPLE 3.9:

Figure 3.13 shows what happens when the *rake* operation is applied to node 1 of the tree of the left-hand side. This operation results in removing nodes 1 and 3, and in making node 4 the parent of node 2, as shown in the right-hand tree. □

Our algorithm for tree contraction uses the *rake* operation as the primitive operation to reduce a given input binary tree into a three-node tree consisting of the root and two leaves. The main technical difficulty lies in avoiding the concurrent *rake* of two leaves whose parents are adjacent. We can avoid that difficulty by a careful application of the *rake* operation to nonconsecutive leaves as they appear from left to right.

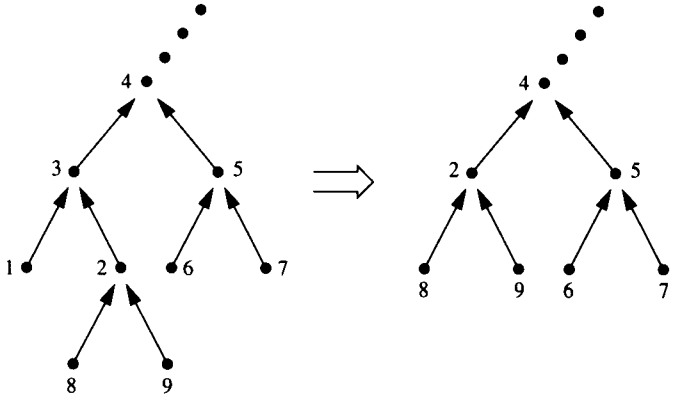


FIGURE 3.13
The process of *raking* a leaf applied to node 1 of the tree on the left-hand side. Nodes 1 and 3 are removed, and node 4 is made the parent of node 2, and the resulting tree is shown on the right-hand side.

For a given array A, A_{odd} is the subarray of A consisting of the odd-indexed elements of A (that is, $a_1, a_3, a_5, \cdots$). We define the subarray A_{even} in a similar fashion. It is easy to check that, once A and its length are given, A_{odd} and A_{even} and their lengths can be determined in $O(1)$ time, using a linear number of operations.

We are ready to present our tree contraction algorithm using the *rake* operation.

ALGORITHM 3.7
(Tree Contraction)

Input: *(1) A rooted binary tree T such that each vertex has exactly two children, and (2) for each vertex v different from the root, the parent p(v) and the sibling* sib*(v).*
Output: *T is contracted to a three-node binary tree.*

begin

1. Label the leaves consecutively in order from left to right, excluding the leftmost and the rightmost leaves, and store the labeled leaves in an array A of size n.

2. **for** $\lceil \log (n + 1) \rceil$ *iterations* **do**

2.1 Apply the rake operation concurrently to all the elements of A_{odd} *that are left children.*

2.2 Apply the rake operation concurrently to the rest of the elements in A_{odd}.
2.3 Set A: = A_{even}.
end

EXAMPLE 3.10:

Consider the tree T given in Fig. 3.14(a). In this case, $A = (1, 2, 3, 4, 5, 6, 7)$, $A_{odd} = (1, 3, 5, 7)$, and $A_{even} = (2, 4, 6)$. During the first iteration of the **for** loop, the *rake* operation is applied to only vertex 3, as specified by step 2.1. This step contracts the given tree to the one shown in Fig. 3.14(b). Step 2.2 of the first iteration results in applying *rake* to vertices 1, 5, and 7, giving us the tree shown in Fig. 3.14(c). At the end of the first iteration, we have $A = (2, 4, 6)$. The second iteration of the loop *rake*s leaves 2 and 6 (step 2.2), resulting in the tree shown in Fig. 3.14(d). Finally, the third iteration shrinks the tree to the three-node tree shown in Fig. 3.14(e). □

Theorem 3.5: *The tree-contraction algorithm (Algorithm 3.7) correctly contracts the input binary tree into a three-node binary tree. This algorithm can be implemented on the EREW PRAM in $O(\log n)$ time, using a linear number of operations, where n is the number of vertices in the input tree.*

Proof: Note that, whenever the *rake* operation is applied concurrently to several leaves, the parents of any two such leaves are not adjacent, and hence *rake* is applied correctly. At the end of each iteration of the main loop, the number of leaves decreases from m to $\lfloor m/2 \rfloor$, where m is the number of leaves at the beginning of the iteration. Hence, after $\lceil \log (n + 1) \rceil$ iterations, all the leaves disappear except the leftmost and rightmost leaves. Therefore, the final tree consists of the root and these two leaves.

Step 1 can be implemented by using the Euler-tour technique. Note that the leaves appear from left to right on the Euler path of T. The *rake* operations are applied in parallel. Hence, steps 2.1 and 2.2 take $O(1)$ time during each iteration. Step 2.3 takes $O(1)$ time as well. The number of operations required by each iteration is $O(|A|)$, where $|A|$ is the current size of the array A. Since $|A_{even}| \leq |A|/2$, the total number of operations needed for the completion of the loop is $O(\sum_i (n/2^i)) = O(n)$. Therefore, the overall time is $O(\log n)$, and the total number of operations is $O(n)$. □

Note that we do not really need to introduce the subarrays A_{odd} and A_{even} in Algorithm 3.7, since the leaves can be stored consecutively in an array, and at each iteration we know the exact positions of the leaves to be removed.

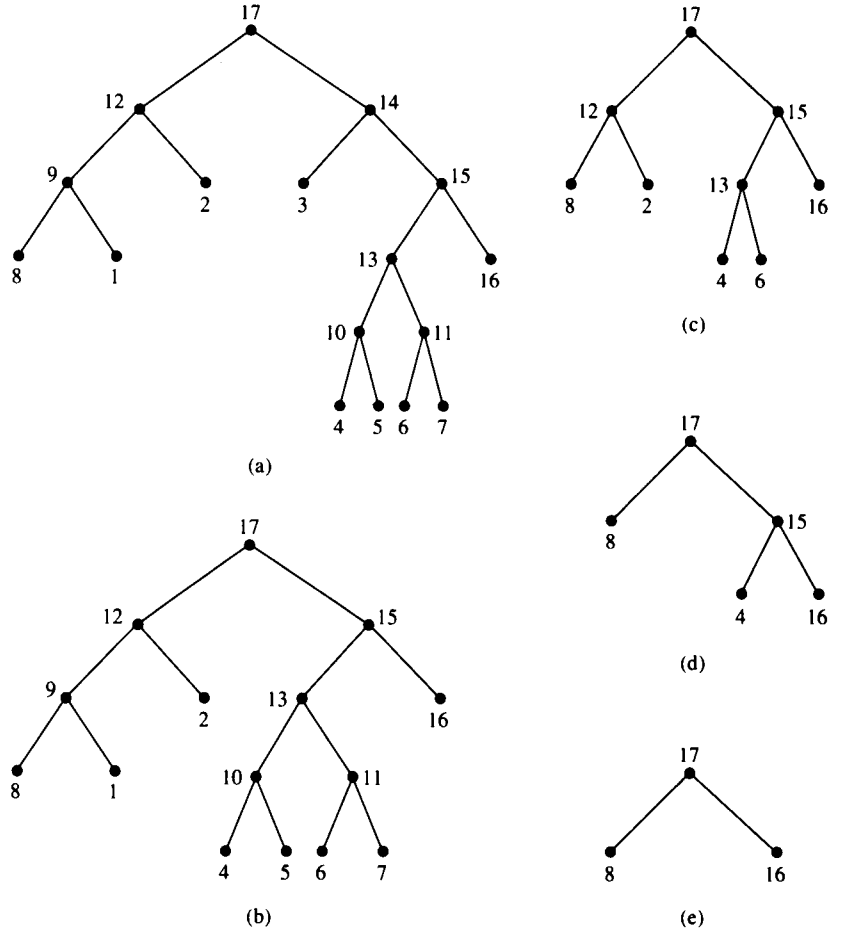


FIGURE 3.14
Tree contraction. (a) Initial tree; (b) tree after *raking* of node 3; (c) tree after *raking* of nodes 1, 5, and 7; (d) tree after *raking* of nodes 2 and 6; (e) the final tree obtained after *raking* of node 4.

3.3.2 EVALUATION OF ARITHMETIC EXPRESSIONS

We now show how to use the tree-contraction algorithm (Algorithm 3.7) for the evaluation of arithmetic expressions. Suppose we are given a rooted binary tree $T = (V, E)$, such that each leaf w contains a constant c_w, and each internal node u contains either the addition operator + or the multiplication

operator $\times$. We assume that each internal vertex has exactly two children. It is simple to modify the algorithm given in this section for the case where some vertices have single children. The **expression evaluation problem** is to determine the value of the expression at the root of T, denoted by $val(T)$.

An intuitive approach for the parallel evaluation of an expression tree is to evaluate each subexpression at an internal node v whose two children are leaves, for all such nodes concurrently. Repeat the process until the value of the root is determined. This approach works well whenever the tree is reasonably well balanced, since many nodes are likely to be eliminated during each stage. On the other hand, very few nodes may be removed during each stage whenever the input tree is long and skinny. In the extreme case, when we have a long chain with leaves attached to it, the previous process requires a linear number of iterations and hence provides no improvement over the sequential algorithm.

To remedy this situation, we relax the constraint that the value of each node has to be fully determined before it can be eliminated. Instead, a node having only a single child that is a leaf can be **partially** evaluated and then removed. To accomplish this partial evaluation, we associate with each node $v \in V$ a label (a_v, b_v) representing the linear expression $a_v X + b_v$, where a_v and b_v are constants, and X is an indeterminate that stands for the possibly unknown value of the subexpression at node v.

During the process of evaluating the expression tree, some nodes are removed and the labels (a_u, b_u) of remaining nodes are adjusted such that the following invariant is maintained.

Invariant (I): Let u be an internal node of the current tree such that u holds the operator $\circ \in \{+, \times\}$ and has the children v and w whose labels are (a_v, b_v) and (a_w, b_w), respectively. Then, the value of the subexpression at u is given by $val(u) = (a_v val(v) + b_v) \circ (a_w val(w) + b_w)$. □

Initially, we assign the label $(1, 0)$ to each node $v \in V$. The invariant (I) holds trivially in this case. We use the tree-contraction algorithm (Algorithm 3.7) to reduce the given input tree T into a three-node tree T' such that $val(T) = val(T')$. We augment the *rake* operation by adjusting the labels (a, b) so as to maintain the invariant **(I)**.

Let v be a leaf and let w be its sibling. When we apply the *rake* operation to v, nodes v and $u = p(v)$ are removed; hence, their contributions to the subexpression computed at node $p(u)$ have to be incorporated into the pair of values (a_w, b_w) stored at w (see Fig. 3.15). Assume, without loss of generality, that v is a left child and that u contains the operator $\circ_u \in \{+, \times\}$. The value of node u is given by $val(u) = (a_v c_v + b_v) \circ_u (a_w X + b_w)$, where X is the unknown value of node w, assuming that the invariant **(I)** holds before the *rake* operation. Hence, the contribution of $val(u)$ to the node $p(u)$ is given by

$E = a_u val(u) + b_u = a_u[(a_v c_v + b_v) \circ_u (a_w X + b_w)] + b_u$, which is a linear expression in X (that is, the unknown value of the subexpression of w) after simplification. Therefore, we can remove v and u, and adjust the pair (a_w, b_w) as implied after simplifying the expression E. The invariant **(I)** is then clearly maintained.

Our algorithm for evaluating arithmetic expressions consists of an initial assignment of the label $(1, 0)$ to each node of the tree, and an application of the tree-contraction algorithm (Algorithm 3.7) such that each *rake* operation is augmented as specified in the previous paragraph. Once the tree-contraction algorithm terminates, we have a three-node tree T' with a root r holding an operator $\circ$ and two leaves u and v containing the constants c_u and c_v. Let (a_u, b_u) and (a_v, b_v) be the labels of u and v, respectively, in the tree T'. Then, by the invariant **(I)** we have $val(T) = val(r) = (a_u c_u + b_u) \circ (a_v c_v + b_v)$. We therefore have the following theorem.

Theorem 3.6: *Given a binary tree T representing an arithmetic expression, the tree-contraction algorithm (Algorithm 3.7) with the augmented rake operation as we have specified can be used to compute $val(T)$ in $O(\log n)$ time, using a linear number of operations.* □

EXAMPLE 3.11:

An expression tree with the initial labels $(1, 0)$ is shown in Fig. 3.16(a). During the first iteration of the **for** loop of the tree-contraction algorithm (Algorithm

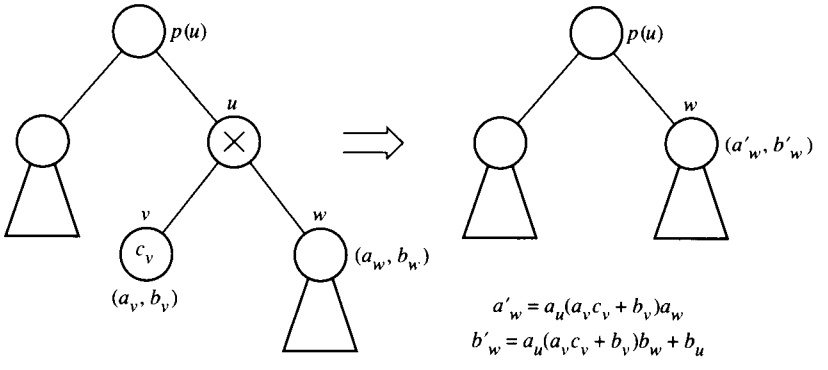


FIGURE 3.15
An example of the *rake* operation applied to node v as needed for evaluating arithmetic expressions, assuming, in this case, that $\circ_u = \times$. Note that a'_w and b'_w are obtained after we have simplified the expression $a_u[(a_v c_v + b_v) \times (a_w X + b_w)] + b_u$.

3.7), the node marked by * is *raked*, thereby generating the tree in Fig. 3.16(b). Note that the label of the node holding -5 changes from $(1, 0)$ to $(1, 2)$. Next, the two starred nodes are *raked*, and the tree is reduced to the one shown in Fig. 3.16(c). During the second iteration, one node is *raked*, resulting in the tree shown in Fig. 3.16(d). Finally during the third iteration, a single node is *raked*, and we end up with the three-node tree shown in Fig. 3.16(e). Consider the new label (4, 14) of the node containing 4. This label is derived from the expression $2[(2X + 10) + (-5 + 2)] + 0 = 4X + 14$. The value of the root expression is given by $(4 \times 4 + 14) + (20 \times 1 + 0) = 50$. □

We can use the same algorithm to compute the values of all the subexpressions by adding a procedure to process the tree in the reverse order by restoring the nodes removed at each iteration and updating the values of their subexpressions. We leave the details to Exercise 3.18.

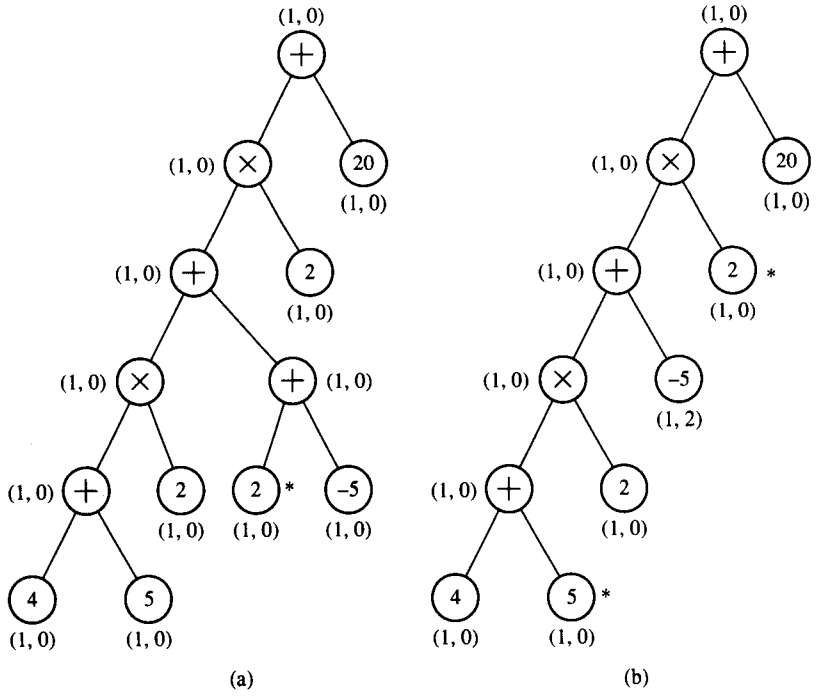


FIGURE 3.16
Expression evaluation by tree contraction for Example 3.11. (a) An initial expression tree. (b) The tree obtained after *raking* of the starred node in (a).

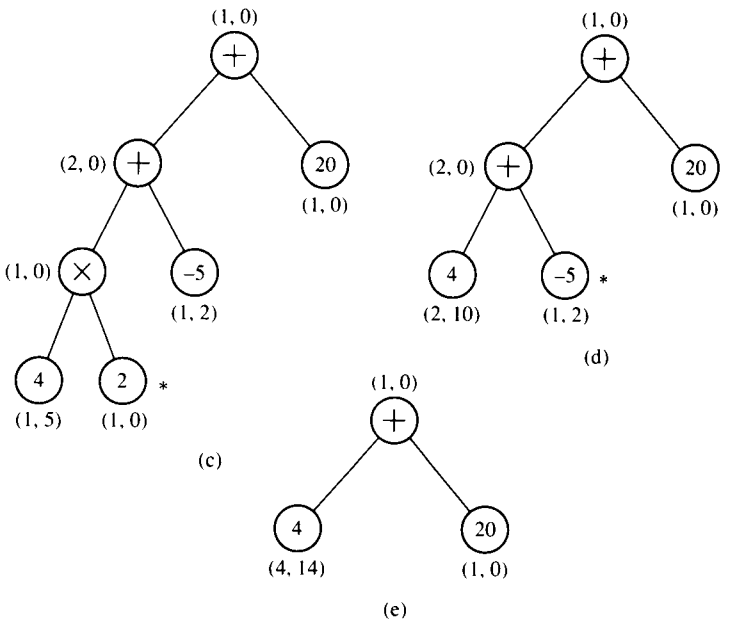


FIGURE 3.16 (*continued*)
(c) The tree generated at the end of the first iteration of the tree-contraction algorithm. (d, e) The trees obtained during the last two iterations.

3.3.3 COMPUTATION OF TREE FUNCTIONS

Let $(U, *)$ be a commutative semigroup with an identity element e. Hence, U is a set of elements with a binary operation $*$, such that $*$ is associative and commutative and $x*e = e*x = x$, for all $x \in U$. Suppose we are given an arbitrary tree $T = (V, E)$, such that each vertex v is labeled with an element $l(v) \in U$. We consider the problem of computing, for each vertex v, the element $L(v) \in U$ that results from applying the operation $*$ to all the elements in the subtree rooted at v.

EXAMPLE 3.12:

Let U be the set of nonnegative integers, and let $*$ be the minimum operator. Add the identity element $+\infty$ to U. Then, $L(v)$ is the minimum element in the subtree rooted at v. □

We outline an $O(\log n)$ time optimal algorithm to solve this problem. We *start by making the tree T binary* as follows. Each vertex v with more than

two children is replaced by a balanced binary tree whose leaves are the children of v. The following example shows how this replacement is done.

EXAMPLE 3.13:

In Fig. 3.17, each of the vertices 1, 2, and 4 has more than two children. Vertex 2 is replaced with a complete binary tree with two additional vertices 2′ and 2″, and each of the vertices 1 and 4 requires the insertion of only one additional vertex. Note that, if we ignore the newly inserted vertices, the set of descendants for each vertex remains the same (although the sibling relationship among the descendants may change). □

Once T has been made binary, we *associate with each* new *internal vertex the label e* (identity element). It is clear that, for each $v \in V$, $L(v)$ has not changed after these modifications.

The problem is now reduced to *evaluating all the subexpressions associated with all the vertices*, where the only operation involved is $*$. This problem is the same as the arithmetic evaluation problem, except that we have only one operation (instead of two). Therefore, we can use the tree-contraction algorithm with the *rake* operation modified appropriately to compute $L(v)$ for each $v \in V$. In particular, this result implies the following corollary.

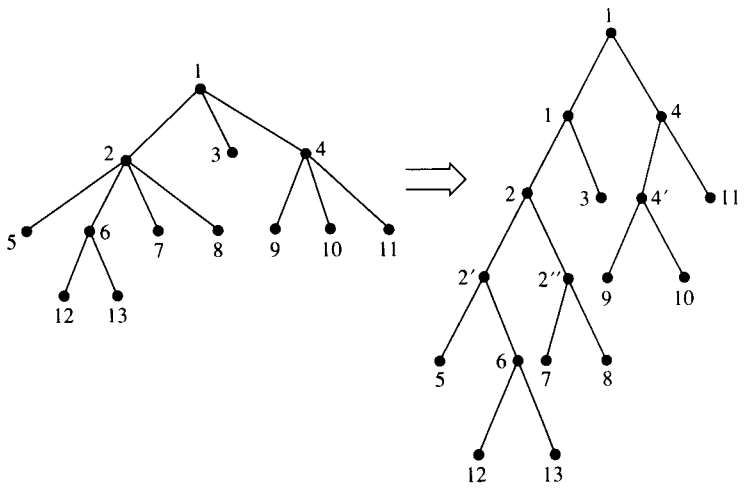


FIGURE 3.17
Process for making an arbitrary tree binary. Each of the vertices 1, 2, and 4 is replaced by a balanced binary tree built on its children.

Corollary 3.1: *Given an arbitrary tree $T = (V, E)$ such that each node is labeled with an element from a linearly ordered set, we can compute the minimum label in the subtree rooted at v, for each vertex $v \in V$, in $O(\log n)$ time, using a linear number of operations.* □

3.4 Lowest Common Ancestors

The **lowest common ancestor** of two vertices u and v of a rooted tree is a node w that is an ancestor to both u and v, and is farthest from the root. For a rooted tree $T = (V, E)$, the problem of finding the *lowest common ancestor* of an arbitrary pair of vertices u and v, denoted by $lca(u, v)$, arises in many situations. In most cases, the pairs (u, v) are determined dynamically during the execution of an algorithm, and, moreover, the number of such pairs may vary considerably. Therefore, we address the problem of *preprocessing T such that a* **query** *$LCA(u, v)$ can be answered very quickly.* In fact, we insist on answering each query in $O(1)$ sequential time. We refer to this problem as the **lowest-common-ancestors (*LCA*) problem.** In this section, we develop an algorithm to solve the *LCA* problem in $O(\log n)$ time, using a linear number of operations.

3.4.1 THE LOWEST-COMMON-ANCESTORS PROBLEM

There are two special cases for which the *LCA* problem can be solved immediately. The first case is when the tree T is a simple path. Computing the distance of each vertex from the root allows us to answer any $LCA(u, v)$ query in constant time by comparing the distances of u and v from the root.

The second case is when T is a complete binary tree. Determining the *inorder number* of each vertex is sufficient to guarantee the handling of each query in constant time as follows. Identify the vertices of T with their inorder numbers. The **inorder traversal** of a binary tree T with root r consists of the inorder traversal of the left subtree of r, followed by r, followed by the inorder traversal of the right subtree of r. Figure 3.18 shows a complete tree whose vertices are identified by their inorder numbers. Then, for any two vertices x and y, $LCA(x, y)$ can be found by the following method. Express x and y as binary numbers and number the bit positions starting from left to right. Let i be the first bit position from the left where x and y disagree; that is, the leftmost $i - 1$ bits $z_1 z_2 \ldots z_{i-1}$ of x and y are identical, and the ith bits are different. Then, $LCA(x, y)$ is equal to the number whose binary representation is given by $z_1 z_2 \ldots z_{i-1}\, 1\, 0 \ldots 0$. The proof of this fact is left to Exercise

3.28, and an example is given next. The fact that the LCA problem can be solved for a complete binary tree with use of the inorder numbers will be assumed for the remainder of this section.

EXAMPLE 3.14:

In Fig. 3.18, consider vertices $9 = (1001)_2$ and $13 = (1101)_2$. Since 9 and 13 disagree on the second leftmost bit, we convert that bit to 1, and convert all subsequent bits to the right of it to 0. Hence $LCA(9, 13) = (1100)_2 = 12$. □

A basic strategy for solving the LCA problem exploits the facts stated in the previous paragraphs by carefully mapping the input tree T into a logarithmic-depth binary tree. References to such work are given at the end of this chapter. Our approach is different and is based on the Euler-tour technique and the range-minima problem (to be introduced shortly).

3.4.2 REDUCTION TO THE RANGE-MINIMA PROBLEM

We start by determining an Euler tour of T. This step assumes that the input tree T is given in a form suitable for the efficient computation of the Euler tour of T, as stated in Section 3.2. We replace each arc $\langle u, v \rangle$ by the vertex v. Then, we define the **Euler array** A to be the corresponding ordered vertices with the root inserted at the very beginning. If $|V| = n$, the size of A is $m = 2n - 1$. From A, we derive array B by taking the level of each element of A as defined in Section 3.2; array B is denoted by $B = level(A)$.

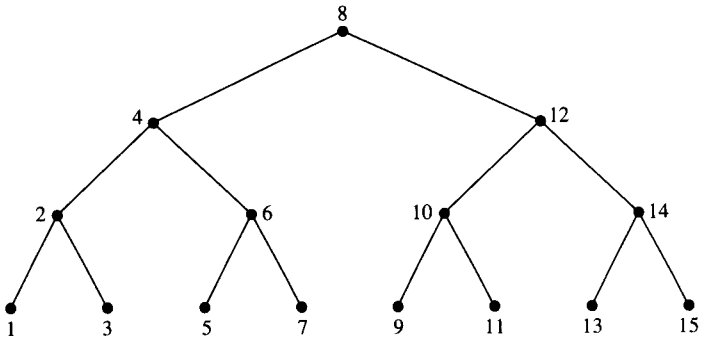


FIGURE 3.18
A complete tree with its vertices identified with their inorder numbers.

EXAMPLE 3.15:

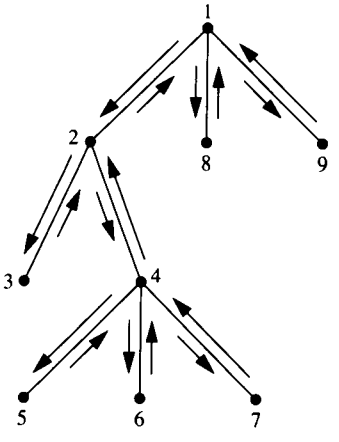
Consider the tree given in Fig. 3.19. The corresponding Euler array A is also shown in the same figure. The array B gives the level of each element of A. □

In addition to A, we need the following information. With each vertex v, we let *level*(v) be the level of v, and $l(v)$ and $r(v)$ be the indices of the *leftmost* and *rightmost* appearances of v in A.

We have already seen, in Section 3.2.2, how to compute *level*(v) by performing a parallel prefix operation on the list defining an Euler path of T. For $v \neq r$, where r is the root of T, $l(v)$ and $r(v)$ can be computed as follows.

Given the array A, the element $a_i = v$ is the leftmost appearance of v in A if and only if $level(a_{i-1}) = level(v) - 1$. Also, $a_i = v$ is the rightmost appearance of v in A if and only if $level(a_{i+1}) = level(v) - 1$. Therefore, $l(v)$ and $r(v)$ can be determined in $O(1)$ time, using a linear number of operations, if A and the level of each element of A are given.

For the remainder of this section, we assume that *level*(v), $l(v)$, and $r(v)$ are known for each vertex v. To answer an LCA query in $O(1)$ sequential time, we can restrict our processing to the array $B = level(A)$ obtained from A by replacing each vertex v by *level*(v). A justification is provided in the next lemma.



$A = (1, 2, 3, 2, 4, 5, 4, 6, 4, 7, 4, 2, 1, 8, 1, 9, 1)$
$B = (0, 1, 2, 1, 2, 3, 2, 3, 2, 3, 2, 1, 0, 1, 0, 1, 0)$

FIGURE 3.19
A tree, the array A of vertices as visited by the Euler tour, and the array $B = level(A)$.

Lemma 3.8: *Given a rooted tree $T = (V, E)$, let A, level(v), $l(v)$, and $r(v)$, for $v \in V$, be as defined previously. Let u and v be two arbitrary distinct vertices of T. Then, the following statements hold:*

1. *u is an ancestor of v if and only if $l(u) < l(v) < r(u)$.*
2. *u and v are not related; that is, u is not a descendant of v and v is not a descendant of u, if and only if either $r(u) < l(v)$ or $r(v) < l(u)$.*
3. *If $r(u) < l(v)$, then $LCA(u, v)$ is the vertex with the minimum level over the interval $[r(u), l(v)]$.*

Proof: We shall prove Claims 1 and 3. Claim 2 can be shown in a similar fashion.

Suppose that u is an ancestor of v. Recall that the Euler tour of T corresponds to a depth-first traversal of T starting from the root. Hence, u is visited before v, and, moreover, the subtree rooted at v is completely visited before the last appearance of v. Hence, $l(u) < l(v) < r(u)$. Conversely, suppose that $l(u) < l(v) < r(u)$, and that u is not an ancestor of v. Since $l(u) < l(v)$, the subtree rooted at u is processed completely before the first visit to v. But then $r(u) < l(v)$, which contradicts the hypothesis. Therefore, u must be an ancestor of v, which proves claim 1.

We now establish claim 3. Suppose that $r(u) < l(v)$. We can easily check that all the vertices whose levels appear in the interval $[r(u), l(v)]$ are either the vertices appearing on the path between u and v (which also includes $LCA(u, v)$), or their descendants. Therefore, the vertex of minimum level must be the LCA of u and v. □

We need an efficient algorithm to preprocess the array $B = level(A)$ so that the following *range-minimum query* can be processed in $O(1)$ sequential time: Given any interval $[k, j]$, where $1 \leq k \leq j \leq n$, find the minimum of $\{b_k, b_{k+1}, \ldots, b_j\}$. This problem, defined for an arbitrary array of elements, is called the **range minima problem**; we discuss it next.

3.4.3 THE RANGE-MINIMA PROBLEM

In the sequential context, the range-minima problem is known to be equivalent to the LCA problem (see the bibliographic notes at the end of this chapter). However, it seems that the range-minima problem can be handled faster than the LCA problem on the PRAM model, unless the Euler tour and the level of each vertex (that is, array B introduced in Section 3.4.1) are given as part of the input to the LCA problem. We first present an optimal $O(\log n)$ time algorithm for the range-minima problem. We then indicate how this

algorithm can be made into an $O(\log \log n)$ time algorithm, using an optimal number of operations.

Basic Range-Minima Algorithm. For clarity, we start by describing a basic preprocessing algorithm that runs in $O(\log n)$ time using $O(n \log n)$ operations.

Let B be the input array of size $n = 2^l$. We wish to preprocess B such that, given any two indices i and j, where $1 \leq i < j \leq n$, we can determine the minimum element in the subarray $\{b_i, b_{i+1}, \ldots, b_j\}$ in $O(1)$ sequential time. Intuitively, it seems that the minima of certain subarrays have to be computed during the preprocessing phase. A natural way to proceed would be to build a complete binary tree on the elements of B such that each internal node v of T holds some information about the array determined by the leaves in the subtree rooted at v. Let us try to determine the information that needs to be stored in each node of T.

Given two indices i and j, we can determine the *LCA* v of the leaves of T holding b_i and b_j in $O(1)$ sequential time, since T is a complete binary tree. If the leaves of the subtree rooted at v correspond exactly to the subarray $\{b_i, b_{i+1}, \ldots, b_j\}$, then it is sufficient to store the minimum element in the node v. However, the subarray associated with v is typically of the form $B_v = \{b_r, \ldots, b_i, \ldots, b_j, \ldots, b_s\}$, where $r \leq i < j \leq s$. In this case, some additional information is needed to answer the range-minimum query. Let u and w be, respectively, the left and the right children of v. Then, the subarrays B_u and B_w associated with u and w partition B_v into, say, $B_u = \{b_r, \ldots, b_i, \ldots, b_p\}$ and $B_w = \{b_{p+1}, \ldots, b_j, \ldots, b_s\}$, for some $i \leq p < j$. The element we seek is the minimum of the following two elements: the minimum of the suffix $\{b_i, \ldots, b_p\}$ of B_u and the minimum of the prefix $\{b_{p+1}, \ldots, b_j\}$ of B_w. Therefore, for each node v, it is sufficient to store the *suffix minima* and the *prefix minima* of the subarray associated with v.

The **prefix minima** of B are the elements of the array $(c_1, c_2, \ldots, c_n)$ such that $c_i = \min\{b_1, \ldots, b_i\}$, for $1 \leq i \leq n$. Similarly, we can define the *suffix minima* of B. Clearly, the prefix minima and the suffix minima of B can be computed in $O(\log n)$ time, using a linear number of operations, by the prefix-sums algorithm.

Our preprocessing algorithm constructs a complete binary tree whose leaves are the elements of B such that each internal node v has associated with it two arrays, P and S, representing, respectively, the prefix minima and the suffix minima of the subarray defined by the leaves of the subtree rooted at v.

ALGORITHM 3.8

(Basic Range Minima)

Input: *An array B of size $n = 2^l$, where l is a positive integer.*

Output: *A complete binary tree with auxiliary variables* $P(h, j)$ *and* $S(h, j)$, $0 \leq h \leq \log n$, $1 \leq j \leq n/2^h$, *such that* $P(h, j)$ *and* $S(h, j)$ *represent, respectively, the prefix and suffix minima of the subarray defined by the leaves of the subtree rooted at* (h, j).

begin

1. **for** $1 \leq j \leq n$ **pardo**
   Set $P(0, j) := B(j)$
   Set $S(0, j) := B(j)$
2. **for** $h = 1$ **to** $\log n$ **do**
   for $1 \leq j \leq n/2^h$ **pardo**
   Merge $P(h - 1, 2j - 1)$ *and* $P(h - 1, 2j)$ *into* $P(h, j)$
   Merge $S(h - 1, 2j - 1)$ *and* $S(h - 1, 2j)$ *into* $S(h, j)$

end

EXAMPLE 3.16:

Consider the array $B = (5, 10, 3, 4, 7, 1, 8, 2)$. The corresponding complete binary tree is shown in Fig. 3.20. The P and S subarrays corresponding to nodes (2, 1), (2, 2), and (3, 1) are the following:

$$P(2, 1) = (5, 5, 3, 3) \qquad P(2, 2) = (7, 1, 1, 1)$$
$$S(2, 1) = (4, 3, 3, 3) \qquad S(2, 2) = (2, 2, 1, 1)$$
$$P(3, 1) = (5, 5, 3, 3, 3, 1, 1, 1) \qquad S(3, 1) = (2, 2, 1, 1, 1, 1, 1, 1)$$

We now explain how these arrays are obtained. We generate the array $P(3, 1)$ by copying $P(2, 1)$ into the first half of $P(3, 1)$, and by copying $P(2, 2)$ into the second half. Then, each element α of the second half of $P(3, 1)$ is replaced by the minimum of α and the last element of $P(2, 1)$—that is, $\min\{\alpha, 3\}$, in this case. The other P or S arrays are generated in the same fashion. □

Suppose that we are now given a range-minimum query over the interval $[i, j]$. Let $v = LCA(i, j)$ in the binary tree constructed previously. Since this binary tree is complete, the node v can be found in $O(1)$ sequential time. Let u and w be the left and right children of v. Then $\min\{a_i, a_{i+1}, \ldots, a_j\}$ is the minimum of two elements: the suffix minimum corresponding to i in the S array of u, and the prefix minimum corresponding to j in the P array of w. Hence, a range-minimum query can be answered in constant time.

EXAMPLE 3.17:

Consider again the array of Fig. 3.20. Suppose that $i = 2$, and $j = 5$, and hence $B(2) = 10$ and $B(5) = 7$. The $LCA(2, 5)$ is the root (3, 1) whose children are the nodes (2, 1) and (2, 2). Note that $S(2, 1) = (4, 3, 3, 3)$ and $P(2, 2) = (7, 1,$

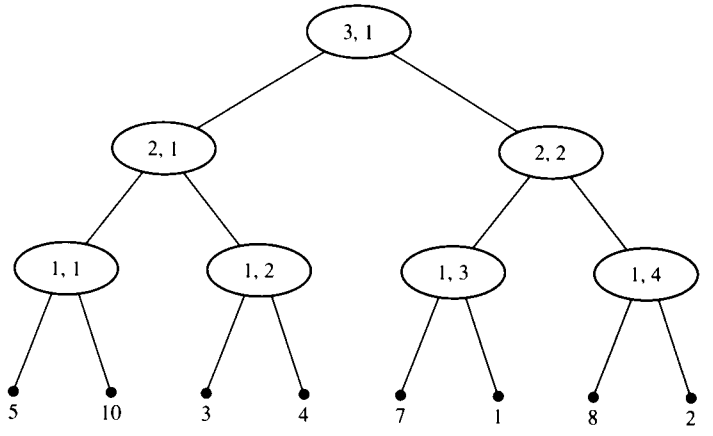


FIGURE 3.20
Binary tree computation for the range-minima problem. Node (h, j) holds the suffix minima and the prefix minima of the subarray associated with the leaves of the subtree rooted at (h, j).

1, 1). Hence, the answer to the query is the minimum of the second element of $S(2, 1)$ and the first element of $P(2, 2)$—that is, $\min\{3, 7\} = 3$. □

Lemma 3.9: *Algorithm 3.8 processes the given input array B in $O(\log n)$ time, using a total of $O(n \log n)$ operations. Each minimum-range query can be answered in $O(1)$ sequential time.*

Proof: The only detail that needs to be justified is how to perform the merging operations of the P and S arrays in $O(1)$ time. Note that the size of each array $P(h,j)$ is equal to 2^h, $1 \leq h \leq \log n$. Merging $P(h-1, 2j-1)$ and $P(h-1, 2j)$ into $P(h,j)$ consists of copying $P(h-1, 2j-1)$ into the first half of $P(h,j)$ and replacing each element α of $P(h-1, 2j)$ with the minimum of α and the last element of $P(h-1, 2j-1)$. Clearly, this merge can be done in $O(1)$ parallel time with $O(2^h = |P(h,j)|)$ operations. The process of merging $S(h-1, 2j-1)$ and $S(h-1, 2j)$ into $S(h,j)$ can be performed in a similar way. Hence, each level of the tree requires $O(1)$ parallel time and $O(2^h \times n/2^h) = O(n)$ operations. Therefore the total running time is $O(\log n)$, and the total number of operations is $O(n \log n)$. □

PRAM Model: The merging operation requires concurrent-read capability, since the last element of $P(h-1, 2j-1)$ has to be compared with each of the

elements in $P(h-1, 2j)$. A similar comment applies to the process of obtaining the array $S(h,j)$. Hence, Algorithm 3.8 runs on the CREW PRAM model. However, we can make the algorithm suitable for the EREW PRAM as follows. Each entry of $P(h,j)$ and $S(h,j)$ will contain the minimum element in the subarray, in addition to the corresponding prefix and to the corresponding suffix minimum. In this case, merging can be completed in $O(1)$ time without any concurrent read. Moreover, the minimum of the newly formed array can be updated without any asymptotic increase in the complexity bounds. The details are left to Exercise 3.29. □

Optimal Basic Range-Minima Algorithm. The basic range minima algorithm (Algorithm 3.8) can be made optimal through use of the standard technique (recall Remark 2.4) as follows:

1. Partition B into equal-sized blocks B_i, each of size $\log n$.
2. Preprocess each B_i block separately for the range-minima problem using an optimal sequential algorithm. Hence, a query whose two elements belong to the same block B_i can now be handled in $O(1)$ sequential time.
3. For each B_i block, compute the minimum element x_i, and its prefix and suffix minima.
4. Preprocess the array $(x_1, x_2, \ldots, x_{n/\log n})$ as in Algorithm 3.8.

Clearly, this preprocessing algorithm takes $O(\log n)$ time, using a linear number of operations.

We claim that this amount of processing is sufficient for answering an arbitrary range-minimum query in constant time.

Suppose we are given a range-minimum query over the interval $[i, j]$, where $1 \le i \le j \le n$. Let i' and j' be the indices of the blocks containing a_i and a_j, respectively. We consider three cases:

1. $i' = j'$. In this case, the range-minimum query is reduced to a query on a single block $B_{i'}$. The optimal sequential preprocessing algorithm of step 2 guarantees an answer within $O(1)$ time.
2. $j' = i' + 1$. The answer is the minimum of two elements: the suffix minimum in block $B_{i'}$ corresponding to a_i, and the prefix minimum in block $B_{j'}$ corresponding to a_j. The prefix minima and the suffix minima of each block are computed in step 3. Hence, the query can be handled in $O(1)$ time in this case.
3. $j' > i' + 1$. In this case, we use the binary tree on the minimum elements x_i's constructed in step 4. We answer the query corresponding to the interval $[i' + 1, j' - 1]$ using the binary tree, as in the previous nonoptimal algorithm. The answer to the query is then the minimum of three elements: the element obtained by using the binary tree, the suffix

minimum corresponding to a_i in $B_{i'}$, and the prefix minimum corresponding to a_j in $B_{j'}$. Hence, the query can be answered in constant sequential time in this case as well.

Theorem 3.7: *The range-minima problem on an array of size n can be solved in $O(\log n)$ time using a total of $O(n)$ operations. Therefore, the LCA problem can also be solved within these bounds. This result guarantees the processing of each query in $O(1)$ sequential time.* □

We can actually design a faster algorithm for the range-minima problem. Using an $O(\log \log n)$ time algorithm to compute the prefix (or suffix) minima of an array of size n (see Exercise 2.19), we build the binary tree of the basic range-minima algorithm (Algorithm 3.8) in $O(\log \log n)$ time using $O(n \log n)$ operations. We can then use the accelerated-cascading strategy to make this algorithm optimal. The details are left to Exercise 3.30.

PRAM Model: The preprocessing algorithm referred to in Theorem 3.7 runs on the EREW PRAM model after Algorithm 3.8 is modified as indicated earlier. The $O(\log \log n)$ time preprocessing algorithm requires the common CRCW PRAM model. □

3.5 Summary

In this chapter, we developed several powerful techniques to deal with basic problems on lists and trees. These techniques include list ranking, the Euler tour, and tree contraction. Optimal $O(\log n)$ time algorithms were then described for computing tree functions, for evaluating arithmetic expressions, and for solving the *LCA* problem. Table 3.1 provides a summary of these results.

Exercises

3.1. Prove Lemma 3.1.

3.2. In Algorithm 3.2, we assumed that the independent set I does not contain the first or the last node of the list. Rewrite Algorithm 3.2 without this restriction.

TABLE 3.1
ALGORITHMS INTRODUCED IN THIS CHAPTER.

Algorithm		Section	Time	Work	PRAM Model
3.1	List Ranking Using Pointer Jumping	3.1.1	$O(\log n)$	$O(n \log n)$	EREW
3.2	Removing Nodes of an Independent Set	3.1.1	$O(\log n)$	$O(n)$	EREW
3.3	Simple Optimal List Ranking	3.1.1	$O(\log n \log\log n)$	$O(n)$	EREW
3.4	Contracting a List (A Single Stage)	3.1.2	$O(1)$	$O\left(\frac{n}{\log n}\right)$	EREW
	List Ranking	3.1.2	$O(\log n)$	$O(n)$	EREW
	Constructing an Euler Tour	3.2.1	$O(1)$	$O(n)$	EREW
3.5	Rooting a Tree	3.2.2	$O(\log n)$	$O(n)$	EREW
3.6	Postorder Numbering	3.2.2	$O(\log n)$	$O(n)$	EREW
	Computing Vertex Level	3.2.2	$O(\log n)$	$O(n)$	EREW
	Computing Number of Descendants	3.2.2	$O(\log n)$	$O(n)$	EREW
3.7	Tree Contraction	3.3.1	$O(\log n)$	$O(n)$	EREW
	Evaluation of Arithmetic Expressions	3.3.2	$O(\log n)$	$O(n)$	EREW
	Computation of Tree Functions	3.3.3	$O(\log n)$	$O(n)$	EREW
3.8	Basic Range Minima	3.4.3	$O(\log n)$	$O(n \log n)$	CREW
	Range Minima	3.4.3	$O(\log n)$	$O(n)$	CREW
	LCA	3.4.2 and 3.4.3	$O(\log n)$	$O(n)$	CREW

3.3. Rewrite Algorithm 3.2 for an arbitrary processor P_i such that P_i stores the information concerning the removed nodes on a stack. Assume that p, the number of processors, divides n evenly. Include a segment that restores the original list and finds the ranks of the deleted nodes.

3.4. Suppose that we use the $O(\log^* n)$ time 3-coloring algorithm (Theorem 2.9) to identify a large independent set needed in Algorithm 3.3. Can you make the resulting list-ranking algorithm run in $O(\log n)$ time with $O(n \log^* n)$ operations? Justify your answer.

3.5. Write the procedure to execute step 2.4 of Algorithm 3.3. Make sure that the names of the nodes are adjusted properly.

3.6. Let $q(n) = 1/\log\log n$. Show that

$$\frac{1}{q(n)}\left(1 - \frac{q(n)}{4}\right)^{c \log n} < (1 - q(n))^{\log n},$$

for any constant $c \geq 5$ and for large enough n. Does the inequality hold if we choose $q(n) = 1/\log n$? *Hint*: Use the fact that $\lim_{x \to \infty} (1 + (\alpha/x))^x = e^{\alpha}$.

3.7. Suppose we are given an array S of size n representing the nodes of a set of linked lists. Each node i has at most one successor whose index is $S(i)$. Show how to rank the nodes within each list optimally—that is, in $O(\log n)$ time, using a linear number of operations.

3.8. Fill in the details required to perform step 4 of Algorithm 3.3. Write down the loop to execute step 4.

3.9. Augment Algorithm 3.4 such that the list can be restored and all the nodes ranked after the contracted list is processed. Show how to restore the list in $O(\log n)$ time, using a linear number of operations.

3.10. You are given a set of linked lists, some of whose nodes are marked with labels from a linearly ordered set. Let n be the total number of nodes. Develop an $O(\log n)$ time EREW PRAM algorithm to find, for each list, the node that is marked with the minimum label. Your algorithm should use $O(n)$ operations.

3.11. Suppose that a tree T is given by its adjacency lists without the additional pointers introduced in Section 3.2. Develop an algorithm to determine the Euler tour. What are the corresponding bounds for the running time and the total number of operations? Can you improve the performance of your algorithm if the degree of each vertex is bounded by a constant?

3.12. Let T be a tree represented by an array such that the children of each internal node occupy consecutive locations of the array and each internal node has the array indices to its first and last child in the array. Using this input representation, show how to compute the Euler tour of T efficiently.

3.13. The preorder traversal of a rooted tree $T = (V, E)$ consists of a traversal of the root r, followed by the preorder traversals of the subtrees of r from left to right. Show how to obtain the preorder number of each vertex v optimally in $O(\log n)$ time on the EREW PRAM model, where $n = |V|$.

3.14. Consider an arbitrary rooted tree $T = (V, E)$ such that, for each vertex v, you are given the next sibling of v, denoted by $s(v)$, and the first child of v, denoted by $fc(v)$. (If no sibling exists, then $s(v) = 0$; if v is a leaf, then $fc(v) = 0$.)

a. Develop an $O(\log n)$ time algorithm to identify, for each vertex v, the parent $p(v)$ of v. Your algorithm must use a linear number of operations.

b. Develop an $O(\log n)$ time algorithm that stores the leaves in consecutive memory locations as they appear in T from left to right. Your algorithm must use a linear number of operations.

3.15. Given a rooted binary tree $T = (V, E)$, with root r, the *inorder traversal* of T consists of the inorder traversal of the left subtree of r, followed by r, followed by the inorder traversal of the right subtree. Develop an $O(\log n)$ time algorithm to assign to each vertex of T its inorder number. Your algorithm should use $O(n)$ operations.

3.16. Let $T = (V, E)$ be a rooted directed tree such that $V = \{1, 2, \ldots, n\}$ and E is represented by a set of adjacency lists that can support the Euler-tour technique. You are given, for each vertex i, a weight $w(i)$. Develop an $O(\log n)$ time algorithm to compute $W(i) = \Sigma w(j)$, where the sum is over all vertices j on the path from i to the root of T. Your algorithm must use a linear number of operations.

Hint: Use a modified version of the pointer jumping technique. Decompose the vertices into bands such that each band contains the vertices between levels $i2^k$ to $(i + 1)2^k - 1$ for the kth iteration. Apply the standard operations to a selected level in each band. After this process is completed, reverse the procedure to compute the W values of the remaining vertices.

3.17. *Let $T = (V, E)$ be a binary tree with $V = \{1, 2, \ldots, n\}$. We are given two arrays P and I such that $P(j)$ and $I(j)$ denote, respectively, the preorder and inorder numbers of vertex j. Develop an $O(\log n)$ time parallel algorithm to determine the left and the right children of each vertex, if they exist.

3.18. Provide an algorithm to determine the subexpressions corresponding to all the vertices of a rooted binary tree representing an arithmetic expression over the operators $\{+, \times\}$. The algorithm described in Section 3.3.2 (Algorithm 3.7 properly modified) evaluates the expression at the root. Modify this algorithm so that the tree could be processed in the reverse order and all the subexpressions could then be evaluated. Your overall algorithm should run in $O(\log n)$ time, using a linear number of operations.

3.19. Modify Algorithm 3.7 so that no auxiliary subarrays will be needed. The resulting algorithm should still run in $O(\log n)$ time, using a linear number of operations.

3.20. You are given a binary tree such that each vertex v contains an item i_v from a linearly ordered set and a pointer $p(v)$ to its parent (if v is the

root, then $p(v) = 0$). Develop an algorithm that, for each vertex v, will compute the minimum item in the subtree rooted at v. Your algorithm must run in $O(\log n)$ time, using $O(n)$ operations.

3.21. Let $S = \{s_1, \cdots, s_k\}$ be a finite set, and let $F = \{f: S \times S \rightarrow S\}$ be the set of binary functions on S. Each such binary function is specified by a $k \times k$ table T_f, where $T_f(i, j)$ contains the element of S corresponding to $f(s_i, s_j)$. Let T be a binary tree such that its leaves are labeled by elements from S, and each internal node u contains a function $f_u \in F$. Develop a parallel algorithm to perform the corresponding algebraic computation on T. State the running time and the total number of operations. Note that a function $f \in F$ is not necessarily commutative or associative.

3.22. *Let $T = (V, E)$ be a binary tree. A subset V' of V is called a *vertex cover* if each edge of T has at least one endpoint in V'. Develop an $O(\log n)$ time parallel algorithm to determine a *minimum* vertex cover. Your algorithm should use a linear number of operations. *Hint:* An optimal sequential algorithm processes the tree bottom-up, by selecting nodes to be or not to be in the minimum cover. The algorithm selects the leaves not to be in the minimum cover, then selects each node connected to a leaf to be in the cover, and so on. Augment the *rake* operation of the tree-contraction algorithm (Algorithm 3.7) to deduce a parallel algorithm.

3.23. Let $T = (V, E)$ be an arbitrary tree. Develop an $O(\log n)$ time algorithm to determine a minimum vertex cover of T using the algorithm that you developed to solve Exercise 3.22. *Hint*: Start by making T binary.

3.24. Let $T = (V, E)$ be an arbitrary tree. A *matching* of T is a subset $M \subseteq E$ such that no two edges of M share a vertex. Develop an $O(\log n)$ time algorithm to compute a maximum matching of T using a linear number of operations. *Hint*: Make T binary and use the algorithm that you developed in Exercise 3.22.

3.25. Let $T = (V, E)$ be a rooted tree with root r, and let $size(v)$ be the number of vertices in the subtree rooted at v. The *centroid level* of a vertex v is given by $clevel(v) = \lceil \log_2 (size(v)) \rceil$.

a. Show that, for any given vertex v, all the vertices with the same centroid level form a set of disjoint paths, one of which contains v.

b. *Develop an optimal EREW PRAM algorithm to decompose T into a set of disjoint paths, each of which is a maximal path with the property that all its vertices have the same centroid level. Each output path should be stored in a consecutive memory location in the same order as in the path.

3.26. Let $T = (V, E)$ be an arbitrary tree with $|V| = n$. A vertex v is a *centroid* of T if removing v generates subtrees each of size less than or equal to $n/2$. Develop an optimal $O(\log n)$ EREW PRAM algorithm to find a centroid of T.

3.27. Let $T = (V, E)$ be a rooted binary tree such that each vertex v has a weight w_v associated with it. Let c be a positive constant. Assume that each internal vertex has exactly degree 2. Develop an optimal algorithm to select a *maximal* set M of edges whose removal generates components such that the total weight of each component is at least c.

3.28. Let T be a complete binary tree whose vertices are identified by their inorder numbers. Let $n = 2^l - 1$ be the number of vertices, and let u be an arbitrary vertex. Suppose that i is the bit position of the rightmost 1, assuming that the positions are numbered starting from 0. (For example, if $u = 8 = 01000$, then $i = 3$.) Show that all the descendants of u have the same $l - i$ leftmost bits. Moreover none of the descendants of u has more than i consecutive rightmost 0 bits. Use these facts to identify uniquely the *LCA* of any two nodes of T.

3.29. Show how to make Algorithm 3.8 an EREW algorithm without increasing the running time or the total number of operations.

3.30. Using an $O(\log \log n)$ time algorithm to compute the prefix (or suffix) minima of an array of size n (see Exercise 2.19), develop an algorithm to solve the range-minima problem in $O(\log \log n)$ time. What is the total number of operations used? Can you make your algorithm optimal? Explain your answer. Specify the PRAM model you need.

3.31. *Given an array of length n containing a legal sequence of parentheses, determine for each left parenthesis in the sequence its matching right parenthesis. Your algorithm should run in $O(\log n)$ time using a total of $O(n)$ operations. *Hint*: Start by assigning a weight of $+1$ to each left parenthesis and a weight of -1 to each right parenthesis. Compute the prefix sums.

3.32. Given a legal arithmetic expression stored in an array of length n, develop an algorithm to find the corresponding binary tree representation. You may assume that the operators appearing in the expression are restricted to $\{+, -, \times\}$. Use the parenthesis-matching algorithm that you developed for Exercise 3.31. Your algorithm should run in $O(\log n)$ time, using a total of $O(n)$ operations.

3.33. Given a tree $T = (V, E)$, such that $|V| = n$, you wish to root T at a vertex r such that the corresponding rooted tree will be of minimum height. Develop an $O(\log n)$ time algorithm to identify such a root. The total number of operations used must be linear. Assume that the tree representation is suitable for the Euler-tour computation.

3.34. Let A be an array $A = (a_1, a_2, \ldots, a_n)$ of distinct elements drawn from a linearly ordered domain. A pair of elements $\langle a_i, a_{i'} \rangle$ is a *matching pair* if $i < i'$ and $a_i < a_{i'}$. You wish to identify a set M of matching pairs such that the elements of A not in any matching pair in M form a strictly decreasing sequence in the order they appear in A.

a. Develop an $O(n)$ sequential time algorithm to obtain such a set M.

b. Show how to obtain such a set in $O(\log n)$ parallel time on the PRAM model. What is the total number of operations used?

Hint: Use a balanced binary tree on the a_i's.

3.35. *Two rooted trees $T = (V, E)$ and $T' = (V', E')$ are *isomorphic* if there exists an isomorphism $f: V \to V'$ that maps the children of a vertex v into the children of $f(v)$. Develop a parallel algorithm to test whether two rooted trees are isomorphic. *Hint*: A linear-time sequential algorithm assigns labels to the vertices level by level using a bucket-sorting algorithm to combine the labels of the children of each vertex. A possible parallel version consists of identifying a centroid of each tree, making a recursive call to each of the resulting subtrees, then combining the labels.

3.36. Let $S = \{S_1, S_2, \ldots, S_n\}$ be a family of sets whose elements are drawn from a universal set U. The *intersection graph* $G_S = (V_S, E_S)$ of S is such that $V_S = \{v_1, \ldots, v_n\}$, each v_i representing S_i, and $(v_i, v_j) \in E_S$ if and only if $S_i \cap S_j \neq \emptyset$. If S is a set of intervals on the real line, then G_S is called an *interval graph*. Develop an algorithm to determine a minimum coloring of an interval graph that runs in $O(\log n)$ time, where n is the number of vertices. What is the total number of operations?

3.37. Let $T = (V, E)$ be an arbitrary tree given by its adjacency lists with the additional pointers to support the Euler-tour construction. Suppose that $|V| = n = 2^k$. Let *preorder*(v) be the preorder number of v. Define $g(v)$ to be the number that has the maximal number of rightmost 0 bits in the closed interval $[preorder(v), preorder(v) + size(v) - 1]$.

a. Show that g partitions T into paths such that each path consists of the vertices with the same g value.

b. Let $T' = (V', E')$ be the complete binary tree on n vertices. Identify each vertex v' with its inorder number. Show that g maps each $v \in V$ into $g(v) \in V'$ such that the descendants of v are mapped into descendants of $g(v)$ in T'.

c. *Show that, for each $v \in V$, the number of the distinct g values of all the ancestors of v is at most $\log n$. Show how to record all these g values using a single string of $O(\log n)$ bits.

d. *Using the facts given, deduce a solution to the *LCA* problem. The corresponding preprocessing algorithm should run in $O(\log n)$ time, using a total of $O(n)$ operations. No concurrent-read or concurrent-write capability is needed.

Bibliographic Notes

The importance of the list-ranking problem seems to have been recognized first by Wyllie [25]. This problem was later addressed by several researchers [23, 16, 24, 6, 8, 2]. The first optimal $O(\log n)$ time algorithm was discovered by Cole and Vishkin [8]. The algorithm described in Section 3.1.2 is from Anderson and Miller [2]. Another simple $O(\log n)$ time optimal algorithm is described in [9] for the CRCW PRAM model. This algorithm is based on an $O(\log n/\log\log n)$ time procedure for computing the prefix sums of a sequence of n numbers, each of which can be represented by $O(\log n)$ bits. A technique similar to the Euler-tour technique was used by Wyllie [25] for tree traversals. Its use as a general technique was described by Tarjan and Vishkin [21]. An early important contribution to the parallel evaluation of arithmetic expressions was given by Brent [5]. The problem was later addressed in the context of the PRAM model by several researchers [18, 12, 1, 15, 7]. Our presentation follows Kosaraju and Delcher [15]. Extensions to more general arithmetic expressions have been given in [22, 17]. The connection between the *LCA* problem and the range-minima problem was initially observed in the sequential context in [11]. This connection and the Euler-tour technique were exploited in [4] to obtain a fast and optimal parallel algorithm. Another optimal parallel algorithm for the *LCA* problem has appeared in [19].

The tree representation sketched in Exercise 3.14 is from [10]. A solution to Exercise 3.16 and several parallel algorithms related to forest and term matching can be found in [14]. An optimal $O(\log\log n)$ time algorithm to solve the problem described in Exercise 3.17, and an optimal $O(\log\log n)$ time algorithm for the parentheses-matching problem (Exercise 3.31), are given in [3]. Solutions to Exercises 3.22 and 3.25 can be found in [7]. Exercise 3.21 is taken from [13]; Exercise 3.27 is taken from [15]. Tree isomorphism (Exercise 3.35) and the general topological matching problem were considered in [20] on a mesh-connected computer. The solution to Exercise 3.37 can be found in [19].

References

1. Abrahamson, K., N. Dadoun, D. A. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10(2):287–302, 1989.
2. Anderson, R., and G. Miller. Deterministic parallel list ranking. In J. Reif, editor, *Proceedings Third Aegean Workshop on Computing, AWOC 88*, Corfu, Greece, 1988, pages 81–90. Springer-Verlag, New York.
3. Berkman, O., B. Schieber, and U. Vishkin. Some doubly logarithmic optimal parallel algorithms based on finding all nearest smaller values. Technical Report UMIACS-TR-88-79, Institute for Advanced Computer Studies, University of Maryland, College Park, MD, 1988.
4. Berkman, O., and U. Vishkin. Recursive star-tree parallel data-structure. Technical Report UMIACS-TR-90-40, Institute for Advanced Computer Studies, University of Maryland, College Park, MD, 1990.
5. Brent, R. P. The parallel evaluation of general arithmetic expressions. *JACM*, 21(2):201–208, 1974.
6. Cole, R., and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986.

7. Cole, R., and U. Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, 3(3):329–346, 1988.
8. Cole, R., and U. Vishkin. Approximate parallel scheduling, Part I: The basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM J. Computing*, 17(1):128–142, 1988.
9. Cole, R., and U. Vishkin. Faster optimal prefix sums and list ranking. *Information and Computation*, 81(3):344–352, 1989.
10. Eppstein, S., and Z. Galil. Parallel algorithmic techniques for combinatorial computation. *Ann. Rev. Comput. Sci.*, 3:233–283, Annual Reviews Inc., Palo Alto, CA, 1988.
11. Gabow, H. N., J. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proceedings Sixteenth Annual ACM Symposium on Theory of Computing*, Washington, DC, 1984, pages 135–143, ACM Press, New York.
12. Gibbons, A., and W. Rytter. An optimal parallel algorithm for dynamic evaluation and its applications. In *Proceedings Sixth Conference on Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science 241*, New Delhi, India, 1986, pages 453–469. Springer-Verlag, New York.
13. He, X., and Y. Yesha. Binary tree algebraic computations and parallel algorithms for simple graphs. *Journal of Algorithms*, 9(1):92–113, 1988.
14. Kedem, Z. M., and K. V. Palem. Optimal parallel algorithms for forest and term matching. *Theoretical Computer Science* (in press).
15. Kosaraju, S. R., and A. Delcher. Optimal parallel evaluation of tree-structured computations by raking. In *Proceedings of AWOC88*, Corfu, Greece, 1988, pages 101–110. Springer-Verlag, New York.
16. Kruskal, C., L. Rudolph, and M. Snir. The power of parallel prefix. *IEEE Transactions on Computers*, C-34(10):965–968, 1985.
17. Miller, G. L., V. Ramachandran, and E. Kaltofen. Efficient parallel evaluation of straight-line code and arithmetic circuits. *SIAM J. Computing*, 17(4):687–695, 1988.
18. Miller, G. L., and J. H. Reif. Parallel tree contraction and its applications. In *Proceedings Twenty-Sixth Annual IEEE Symposium on Foundations of Computer Science*, Portland, OR, 1985, pages 478–489. IEEE Press, Piscataway, NJ.
19. Schieber, B., and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Computing*, 17(6):1253–1262, 1988.
20. Stout, Q. F. Topological matching. In *Proceedings Fifteenth Annual ACM Symposium on Theory of Computing*, Boston, MA, 1983, pages 24–31. ACM Press, New York.
21. Tarjan, R. E., and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Computing*, 14(4):862–874, 1985.
22. Valiant, L. G., S. Skyum, S. Berkowitz, and C. Rackoff. Fast parallel computation of polynomials using few processors. *SIAM J. Computing*, 12(4):641–644, 1983.
23. Vishkin, U. Randomized speed-ups in parallel computation. In *Proceedings Sixteenth ACM Symposium on Theory of Computing*, Washington, D.C., 1984, pages 230–239. ACM Press, New York.
24. Wagner, W., and Y. Han. Parallel algorithms for bucket sorting and the data dependent prefix problem. In *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL, 1986, pages 924–930.
25. Wyllie, J. C. *The Complexity of Parallel Computations.* PhD thesis, Computer Science Department, Cornell University, Ithaca, NY, 1979.

4

Searching, Merging, and Sorting

The task of processing a table consisting of records whose keys come from a linearly ordered set arises in many practical situations. Examples of specific processing tasks include searching for a key in the table and sorting the keys of the table. Such tasks typically involve repeated applications of comparison and data-movement operations. We assume that each key is an atomic unit that cannot be manipulated as an integer or as a string of bits. This class of problems has a rich mathematical theory that still attracts a substantial research interest.

In this chapter, optimal algorithms for basic problems in searching, merging, and sorting are presented. Several algorithmic techniques are explored, including *parallel searching* (Section 4.1), *partitioning* (Section 4.2), *pipelined* or *cascading divide-and-conquer* (Section 4.3), and *bitonic sorting* (Section 4.4). These techniques are used to develop efficient parallel algorithms for searching, merging, selection, and sorting. In addition, we introduce for the first time in this book lower-bound proofs on the parallel complexity of certain problems (Section 4.6). These proofs are especially tailored for the class of comparison problems considered in this chapter.

4.1 Searching

Let $X = (x_1, x_2, \ldots, x_n)$ be n distinct elements drawn from a linearly ordered set $(S, \leq)$ such that $x_1 < x_2 < \cdots < x_n$. Given an element $y \in S$, we are interested in solving the following **search problem:** identify the index i for which $x_i \leq y < x_{i+1}$, where $x_0 = -\infty$ and $x_{n+1} = +\infty$, and $-\infty$ and $+\infty$ are two elements that are added to S and that satisfy $-\infty < x < +\infty$, for all $x \in S$.

The well-known binary search method solves this problem in $O(\log n)$ optimal sequential time. This method consists of comparing y with the middle element of X. Based on the outcome of this comparison, the search either is terminated with a success or it can be restricted to the upper or the lower half of X. The process is repeated until either an element x_i is encountered such that $y = x_i$ or the size of the subarray under consideration is equal to 1. Hence, in either case, the solution is determined.

A natural extension of the binary search method to parallel processing is **parallel searching,** in which we compare y concurrently with several elements of X, say p elements of X, which split X into $p + 1$ segments of almost equal lengths. The outcome of this parallel comparison step is either to identify an element x_i that is equal to y, or to restrict the search to one of the $p + 1$ segments. We repeat this process until either an element x_i is encountered such that $y = x_i$ or the number t of elements in the subarray under consideration is no more than p. In the former case, the solution is found; in the latter case, the solution can be determined by t comparisons being made in parallel.

In this section, we shall not use the WT framework to present and analyze our parallel-search algorithm. Instead, the algorithm is couched within the alternative time-processors framework. The parameter p that we introduced can be viewed as the number of processors available on our PRAM model.

The parallel-search algorithm for processor P_j is given next, where $1 \leq j \leq p$. Processor P_1 is responsible for various initializations and for taking care of the boundary cases. The variable c_j is used to indicate the outcome of the comparison performed in a given step; the indices l and r are pointers to the (left and right) boundaries of the current subarray under consideration.

ALGORITHM 4.1
(Parallel Search for Processor P_j)
Input: *(1) An array* $X = (x_1, x_2, \ldots, x_n)$ *such that* $x_1 < x_2 < \ldots < x_n$*; (2) an element* y*; (3) the number* p *of processors, where* $p \leq n$*; (4) the processor number* j*, where* $1 \leq j \leq p$.
Output: *An index* i *such that* $x_i \leq y < x_{i+1}$.

begin

1. **if** $(j = 1)$ **then do**

1.1. Set $l := 0; r := n + 1; x_0 := -\infty; x_{n+1} := +\infty$

1.2. Set $c_0 := 0; c_{p+1} := 1$

2. **while** $(r - l > p)$ **do**

2.1. **if** $(j = 1)$ **then** $\{$*set* $q_0 := l; q_{p+1} := r\}$

2.2. Set $q_j := l + j \lfloor \frac{r-l}{p+1} \rfloor$

2.3. **if** $(y = x_{q_j})$ **then** $\{$**return**(q_j); **exit**$\}$

else $\{$*set* $c_j := 0$ *if* $y > x_{q_j}$ *and* $c_j := 1$ *if* $y < x_{q_j}\}$

2.4. **if** $(c_j < c_{j+1})$ **then** $\{$*set* $l := q_j; r := q_{j+1}\}$

2.5. **if** $(j = 1$ *and* $c_0 < c_1)$ **then** $\{$*set* $l := q_0; r := q_1\}$

3. **if** $(j \leq r - l)$ **then do**

3.1. Case statement:

$y = x_{l+j}$: {**return** $(l + j)$; **exit**}

$y > x_{l+j}$: set $c_j := 0$

$y < x_{l+j}$: set $c_j := 1$

3.2. **if** $(c_{j-1} < c_j)$ **then return** $(l + j - 1)$

end

EXAMPLE 4.1:

Let X be the array $X = (2, 4, 6, \cdots, 30)$ consisting of all even integers between 2 and 30, and let $y = 19$. Suppose that $p = 2$. After the execution of step 1, P_1 will set $l = 0, r = 16, c_0 = 0, c_3 = 1, x_0 = -\infty$, and $x_{16} = +\infty$. The **while** loop runs for three iterations; the effect of each such iteration is shown in the following table:

iteration	1	2	3
q_0	0	5	7
q_1	5	6	8
q_2	10	7	9
q_3	16	10	10
c_0	0	0	0
c_1	0	0	0
c_2	1	0	0
c_3	1	1	1
l	5	7	9
r	10	10	10

During the execution of step 3.1, P_1 sets $c_1 = 1$. Hence, at step 3.2, P_1 verifies that $c_0 < c_1$ and returns the index 9. □

We are ready for the following theorem.

Theorem 4.1: *Given an array* $X = (x_1, x_2, \ldots, x_n)$ *with* $x_1 < x_2 < \ldots < x_n$ *and an element* y, *Algorithm 4.1 determines the index* i *such that* $x_i \leq y < x_{i+1}$. *The parallel time required is* $O\left(\frac{\log (n+1)}{\log (p+1)}\right)$, *where* p *is the number of processors used.*

Proof: The correctness proof is simple and is left to the reader (see Exercise 4.2).

As for the number of steps, notice that, after the ith iteration of the **while** loop, the size of the subarray to be searched is reduced from $s_i = r - l$ to $s_{i+1} \leq \frac{r-l}{p+1} + p = \frac{s_i}{p+1} + p$, which is the maximum possible length of the $(p+1)$st segment. Setting $s_0 = n + 1$, it is straightforward to check that $s_i \leq \frac{n+1}{(p+1)^i} + p + 1$ satisfies this recurrence. Therefore, the number of iterations needed is $O\left(\frac{\log(n+1)}{\log(p+1)}\right)$, and each iteration takes $O(1)$ time. Hence, the **while** loop requires $O(\log (n+1)/\log (p+1))$ time. Since step 3 takes $O(1)$ time, the time bound stated in the theorem follows. □

PRAM Model: Algorithm 4.1 can be implemented on the CREW PRAM with p processors. A concurrent-read capability is needed, since l, r and the search key y need to be accessed by all the processors. Surprisingly, we can show that $\Omega(\log n - \log p)$ parallel steps are required on the EREW PRAM model even if the search key is made available to all the processors (see Section 10.3). Hence, no significant speedup can be achieved on the EREW PRAM model for any number p of processors such that $p \leq n^c$, where c is any constant satisfying $0 < c < 1$. □

Remark 4.1: According to our notion of optimality, a parallel-search algorithm is optimal if its total number of operations is asymptotically equal to the sequential complexity $\Theta(\log n)$ of the search problem. Hence, Theorem 4.1 shows that optimality on the CREW PRAM is achieved only when p is constant and hence $T(n) = \Theta(\log n)$. However, we show in Section 4.6 that the performance of Algorithm 4.1 cannot be improved. □

4.2 Merging

We have already considered the problem of merging two sorted sequences in Section 2.4. Based on the **partitioning strategy,** an $O(\log n)$-time parallel algorithm was presented there. The corresponding total work is $O(n)$; hence, the algorithm is optimal.

In this section, we refine the previous partitioning strategy to obtain an optimal $O(\log \log n)$ time algorithm that runs on the CREW PRAM. This problem is one of the few known to have such a fast algorithm on the CREW PRAM. Compare the merging problem with the simple problem of computing the maximum of n elements, which requires $\Omega(\log n)$ parallel steps on the CREW PRAM, regardless of the number of the processors available. Therefore, the existence of an optimal $O(\log \log n)$ time merging algorithm that runs on the CREW PRAM is perhaps surprising.

We start by recalling a few definitions presented in Section 2.4. The **rank** of an element x in a given sequence X, denoted by $rank(x : X)$, is the number of elements of X that are less than or equal to x. If X is sorted, it is useful to define the **predecessor** of an arbitrary element x to be the element x_r of X such that $r = rank(x : X)$. **Ranking** a sequence $Y = (y_1, y_2, \ldots, y_m)$ in X amounts to computing the integer array $rank(Y : X) = (r_1, r_2, \ldots, r_m)$, where $r_i = rank(y_i : X)$.

4.2.1 RANKING A SHORT SEQUENCE IN A SORTED SEQUENCE

Let X be a sorted sequence with n distinct elements, and let Y be an arbitrary sequence of size m such that $m = O(n^s)$, where s is a constant that satisfies $0 < s < 1$. The parallel-search algorithm (Algorithm 4.1) can be used to rank each element of Y in X separately. We set the number p of processors of this algorithm to $p = \lfloor n/m \rfloor = \Omega(n^{1-s})$. Then, each element of Y can be ranked in X in $O(\log (n + 1)/\log (p + 1)) = O(1)$ time. The total number of operations used to rank each such element is $O(n/m)$, since $p = \lfloor n/m \rfloor$, and the running time is $O(1)$. We therefore have the following lemma.

Lemma 4.1: *Let Y be an arbitrary sequence with m elements, and let X be a sorted sequence with n distinct elements such that $m = O(n^s)$ for some constant $0 < s < 1$. Then, all the elements of Y can be ranked in X in $O(1)$ time using a total of $O(n)$ operations.* □

PRAM Model: The parallel-search algorithm (Algorithm 4.1) requires a concurrent-read capability. Hence, the procedure referred to in Lemma 4.1 requires the CREW PRAM model. □

4.2.2 A FAST MERGING ALGORITHM

Consider the problem of determining $rank(B : A)$ for sorted sequences A and B of lengths n and m, respectively. Assume that all the elements of A and B are distinct and hence that no element of A appears in B.

As in Section 2.4, we use the partitioning strategy to merge the two sequences A and B. We rank a set of $\sqrt{m}$ elements of B that partition B into blocks of almost equal lengths in the sorted sequence A. The computed ranks of the chosen elements will induce a partition on A into blocks such that each block of A has to fit between two of the chosen elements of B. Hence, the overall problem is now reduced to ranking the elements of each block of B (excluding already ranked elements) into a corresponding block of A.

The algorithm is given next. Figure 4.1 illustrates the partitions introduced in the algorithm.

ALGORITHM 4.2

(Ranking a Sorted Sequence in Another Sorted Sequence)

Input: *Two arrays* $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_m)$ *in increasing order. Assume that* $\sqrt{m}$ *is an integer; otherwise, replace* $\sqrt{m}$ *whenever it occurs by* $\lfloor \sqrt{m} \rfloor$.

Output: *The array* $rank(B : A)$.

begin

1. If $m < 4$, *then rank the elements of* B *by applying the parallel-search algorithm (Algorithm 4.1) with* $p = n$, *and* **exit.**

2. Concurrently rank the elements $b_{\sqrt{m}}, b_{2\sqrt{m}}, \ldots, b_{i\sqrt{m}}, \ldots, b_m$ *in* A *by using the parallel-search algorithm (Algorithm 4.1) with* $p = \sqrt{n}$. *Let* $rank(b_{i\sqrt{m}} : A) = j(i)$, *for* $1 \leq i \leq \sqrt{m}$. *Set* $j(0) = 0$.

3. For $0 \leq i \leq \sqrt{m} - 1$, *let* $B_i = (b_{i\sqrt{m}+1}, \ldots, b_{(i+1)\sqrt{m}-1})$ *and let* $A_i = (a_{j(i)+1}, \ldots, a_{j(i+1)})$; if $j(i) = j(i+1)$, *then set* $rank(B_i : A_i) = (0, \ldots, 0)$, *else recursively compute* $rank(B_i : A_i)$.

4. Let $1 \leq k \leq m$ *be an arbitrary index that is not a multiple of* $\sqrt{m}$, *and let* $i = \lfloor \frac{k}{\sqrt{m}} \rfloor$. *Then* $rank(b_k : A) = j(i) + rank(b_k : A_i)$.

end

EXAMPLE 4.2:

Let $A = (-5, 0, 3, 4, 17, 18, 24, 28)$ and $B = (1, 2, 15, 21)$. At the termination of step 2, we obtain $j(0) = 0, j(1) = 2$ and $j(2) = 6$, where 2 and 6 are the ranks of $b_2 = 2$ and $b_4 = 21$, respectively. During the execution of step 3, we introduce $B_0 = (1)$ and $A_0 = (-5, 0)$, $B_1 = (15)$ and $A_1 = (3, 4, 17, 18)$. In this case, $rank(1 : A_0) = 2$ and $rank(15 : A_1) = 2$. At step 4, we adjust the ranks of $b_1 = 1$ and $b_3 = 15$ as follows: $rank(1 : A) = j(0) + rank(1 : A_0) = 2$ and $rank(15 : A) = j(1) + rank(15, A_1) = 4$. Therefore, we get the array $rank(B : A) = (2, 2, 4, 6)$. □

Lemma 4.2: *Let* A *and* B *be two sorted sequences such that* $|A| = n$ *and* $|B| = m$. *Then, Algorithm 4.2 computes the array* $rank(B : A)$ *in* $O(\log \log m)$ *time using* $O((n + m) \log \log m)$ *operations.*

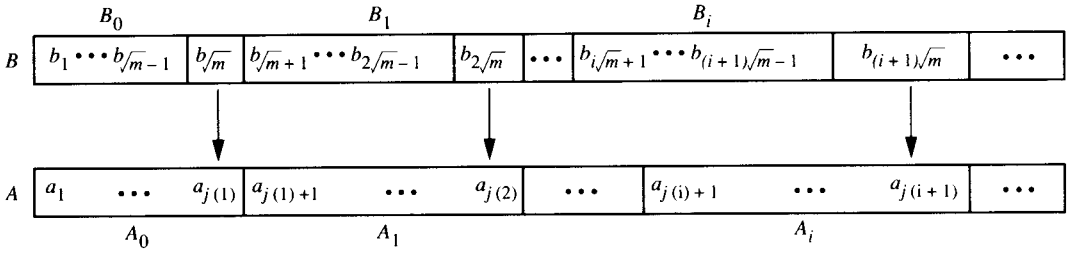


FIGURE 4.1
Partitions induced by Algorithm 4.2. Each block B_i is of size $\sqrt{m}$ but the size of each A_i block may vary. The index $j(i)$ is defined by $j(i) = rank(b_{i\sqrt{m}} : A)$.

Proof: The correctness proof will be by induction on m.

The base case $m = 3$ corresponds to ranking a sequence (b_1, b_2, b_3) in A. Step 1 settles this case.

Suppose that the induction hypothesis holds for all $m' < m$, where $m \geq 4$. We establish the fact that all the elements in B_i are strictly between $a_{j(i)}$ and $a_{j(i+1)+1}$, for any i such that $0 \leq i \leq \sqrt{m} - 1$.

Any element p in B_i satisfies $b_{i\sqrt{m}} < p < b_{(i+1)\sqrt{m}}$. Since $j(i) = rank(b_{i\sqrt{m}} : A)$ and $j(i+1) = rank(b_{(i+1)\sqrt{m}} : A)$, we have $a_{j(i)} < b_{i\sqrt{m}}$ and $b_{(i+1)\sqrt{m}} < a_{j(i+1)+1}$, and thus $a_{j(i)} < p < a_{j(i+1)+1}$. This result implies that any element p of block B_i fits somewhere in block A_i and therefore $rank(p : A) = j(i) + rank(p : A_i)$, since $j(i)$ is the number of elements in A preceding A_i. Hence the correctness proof follows by induction.

We now establish the complexity bounds. Let $T(n, m)$ be the parallel time it takes to rank B in A, where $|B| = m$ and $|A| = n$.

Step 2 invokes $\sqrt{m}$ calls to the parallel-search algorithm (Algorithm 4.1) with $p = \sqrt{n}$. The running time is $O(\log (n+1)/\log (\sqrt{n}+1)) = O(1)$, and the total number of operations required is $O(\sqrt{m}.\sqrt{n}) = O(n + m)$ (since $2\sqrt{m}.\sqrt{n} \leq n + m$). Except for the recursive calls, steps 3 and 4 trivially take $O(1)$ time, with $O(n + m)$ operations.

Let $|A_i| = n_i$, for $0 \leq i \leq \sqrt{m} - 1$. The recursive call corresponding to the pair (B_i, A_i) takes $T(n_i, \sqrt{m})$ time. Hence, $T(n, m) \leq \max_i T(n_i, \sqrt{m}) + O(1)$, and $T(n, 3) = O(1)$. A solution to this recurrence is given by $T(n, m) = O(\log \log m)$. Since the number of operations required for each set of recursive calls is $O(n + m)$, the total number of operations used by Algorithm 4.2 is thus $O((n + m) \log \log m)$. □

PRAM Model: Algorithm 4.2 requires the CREW PRAM model because the parallel-search algorithm requires that model. □

Remark 4.2: Algorithm 4.2 is given in the WT presentation framework. The corresponding processor allocation problem is not straightforward. The reader is asked to provide the details in Exercise 4.11. □

As indicated in Section 2.4, we can solve the problem of merging two sorted sequences A and B by computing the two arrays $rank(A : B)$ and $rank(B : A)$. Therefore, the following result follows immediately from Lemma 4.3.

Corollary 4.1: *Let A and B be two sorted sequences, each of length n. Merging A and B can be done in $O(\log \log n)$ time, using a total of $O(n \log \log n)$ operations.* □

4.2.3 AN OPTIMAL FAST MERGING ALGORITHM

The previous fast merging algorithm (Algorithm 4.2) is clearly nonoptimal. As is usually the case, the fast nonoptimal algorithm is used to solve a suitably chosen reduced-sized version generated by an optimal (and slow) algorithm.

Assume for the remainder of this section that $m = n$. The details for the more general case are left to Exercise 4.12.

We begin by partitioning A and B into blocks, each of size less than or equal to $\lceil \log \log n \rceil$, say $A = (A_1, A_2, \ldots)$ and $B = (B_1, B_2, \ldots)$. We then let $A' = (p_1, p_2, \ldots)$, where p_i is the first element of block A_i of A and $B' = (q_1, q_2, \ldots)$, where q_i is the first element of block B_i of B. As a result, $|A'| = O(n/\log \log n)$ and $|B'| = O(n/\log \log n)$. Then, we rank the elements of A' in B and the elements of B' in A. The merging problem is now reduced to merging nonoverlapping pairs of subsequences; each subsequence is of length $O(\log \log n)$. The details are as follows:

1. Merge $A' = (p_1, p_2, \ldots)$ and $B' = (q_1, q_2, \ldots)$ using the fast nonoptimal algorithm (Algorithm 4.2).

   Explanation: Algorithm 4.2 can be used to perform the merging of A' and B' in $O(\log \log n)$ time using a total of $O(n)$ operations. At this point, we have computed the two arrays $rank(A' : B')$ and $rank(B' : A')$.

2. Determine the two arrays $rank(A' : B)$ and $rank(B' : A)$.

   Explanation: Let $rank(p_i : B') = r_i$; hence p_i must fit somewhere in block B_{r_i} of B since $q_{r_i} < p_i < q_{r_i+1}$ (see Fig. 4.2). For each p_i, we can determine its exact location in B_{r_i} by using a simple sequential algorithm (or binary search algorithm). This algorithm takes $O(\log \log n)$ sequential time per element. Since we have $\leq \lceil n/\log \log n \rceil$ such elements, the array $rank(A' : B)$ can be determined in $O(\log \log n)$ parallel time, using a total of $O(n)$ operations. Similarly, we can obtain $rank(B' : A)$ in $O(\log \log n)$ time, using a total of $O(n)$ operations.

3. For each i, determine the ranks of the elements $A_i - \{p_i\}$ in B, and, for each k, the ranks of the elements $B_k - \{q_k\}$ in A.

   Explanation: We computed $rank(A' : B)$ and $rank(B' : A)$ by the end of step 2. Let $rank(p_i : B) = j(i)$ and $rank(q_k : A) = \tilde{j}(k)$. Recall that p_i is the first element of block A_i and q_k is the first element of block B_k, where each A_i and each B_k is of size less than or equal to $\lceil \log \log n \rceil$. In step 3, we determine the ranks of the remaining elements in each A_i and B_k.

Given the facts that $rank(p_i : B) = j(i)$ and that p_i is the first element of A_i, all the elements of A_i must lie between $b_{j(i)}$ and $b_{j(i+1)+1}$. Hence, if $j(i) = j(i+1)$, each element of A_i has rank $j(i)$ in B.

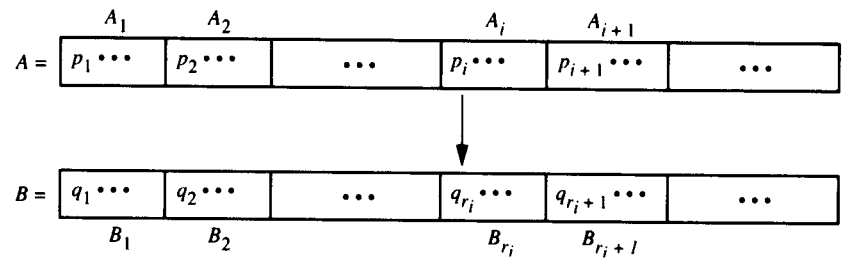


FIGURE 4.2
Step 2 of the optimal merging procedure. Since $rank(p_i : B') = r_i$, where $B' = (q_1, q_2, \dots)$, we have that $q_{r_i} < p_i < q_{r_i+1}$; hence, p_i must fit somewhere in block B_{r_i}.

Suppose that $j(i + 1) > j(i)$. In this case, all the elements $b_{j(i)+1}, \dots, b_{j(i+1)}$ fit between p_i and p_{i+1}. If the number of such elements (that is, the $j(i + 1) - j(i)$ elements $b_{j(i)+1}, \dots, b_{j(i+1)}$) is less than $\log \log n$, then we can merge A_i with the corresponding subarray of B in $O(\log \log n)$ sequential time. Otherwise, there exist elements of B'—say, $q_k, q_{k+1}, \dots, q_{k+s}$—that lie between $b_{j(i)+1}$ and $b_{j(i+1)}$ (see Fig. 4.3). Since the ranks $\tilde{j}(k), \tilde{j}(k + 1), \dots, \tilde{j}(k + s)$ of $q_k, q_{k+1}, \dots, q_{k+s}$ are already known, our problem is reduced to merging no more than $s + 2$ pairs of subsequences, each pair containing $O(\log \log n)$ elements (see Fig. 4.3). Note that the subsequences are completely disjoint. Hence, this merging can be done in $O(\log \log n)$ time, using a linear number of operations (in the total size of the subsequences involved). This process can be performed concurrently for all blocks A_i.

In summary, the optimal merging algorithm consists of three main steps:

1. Partition A and B into blocks $A = (A_1, A_2, \cdots)$ and $B = (B_1, B_2, \cdots)$ such that each block is of size $\leq \lceil \log \log n \rceil$. Use the fast but nonoptimal algorithm (Algorithm 4.2) to merge $A' = (p_1, p_2, \dots)$ and $B' = (q_1, q_2, \dots)$, where p_i and q_i are the first elements of A_i and B_i respectively.
2. Each p_i can now be located in a block B_{r_i}; hence, its exact rank in B can be determined easily, since $|B_{r_i}| \leq \lceil \log \log n \rceil$. The ranks of all the p_i's in B are determined concurrently. Similarly the ranks of the q_i's in A are determined concurrently.
3. Since the exact ranks of the elements of A' in B and the exact ranks of B' in A are known, the merging problem is reduced to a set of nonoverlapping merging subproblems; each of the corresponding pairs of subsequences involves $O(\log \log n)$ elements. All the merging subproblems can now be solved concurrently in $O(\log \log n)$ time, using a linear number of operations.

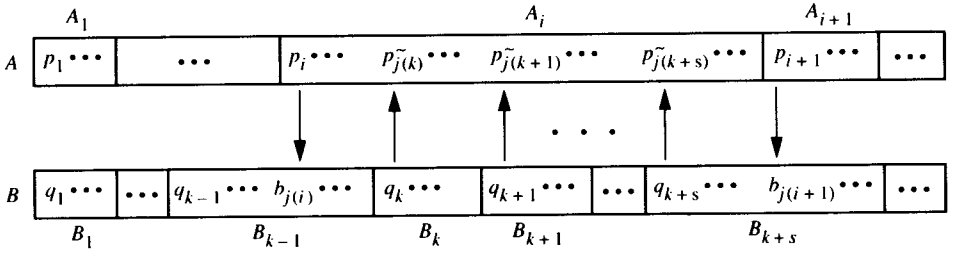


FIGURE 4.3
Step 3 of the optimal merging algorithm. Each A_i and B_k block is approximately of size log log n. The case depicted corresponds to $j(i + 1) - j(i)$ being much larger than log log n. An arrow, say from p_i to $b_{j(i)}$, indicates that p_i must fit after $b_{j(i)}$ in the sequence B (more precisely, $j(i) = rank(p_i : B)$). Any consecutive pair of arrows determines two subsequences that have to be merged.

Therefore we have the following theorem.

Theorem 4.2: *The problem of merging two sorted sequences each of length n can be done in $O(\log \log n)$ time, using a total of $O(n)$ operations.*

PRAM Model: A concurrent-read capability is required by the optimal fast merging algorithm since, for example, it uses the fast nonoptimal merging algorithm (Algorithm 4.2), which requires concurrent-read operations. However, no concurrent write was used, and therefore the optimal $O(\log \log n)$ time algorithm can be implemented on the CREW PRAM model. □

As we shall show in Section 4.6, $\Omega(\log \log n)$ parallel steps are required to merge two sorted sequences each of length n, even on any CRCW PRAM model with $n \log^{\alpha} n$ processors, for any constant α. Hence, the optimal time fast merging algorithm cannot be improved as long as we insist on optimality; that is, it is **work-time (WT) optimal.**

If we are provided with additional information, however, we may be able to perform merging faster. We now discuss a case which pertains to the $O(\log n)$ time sorting algorithm described in Section 4.3.2.

4.2.4 *MERGING WITH THE HELP OF A COVER

Let c be a positive integer. A sorted sequence X will be called a **c-cover** of another sorted sequence Y if Y has at most c elements between each pair of *consecutive* elements in $X_\infty = (-\infty, X, +\infty)$. More precisely, given any two consecutive elements α and β of X_∞, then the set $\{y_i \mid y_i \in Y \text{ and } \alpha < y_i \leq \beta\}$ has at most c elements.

EXAMPLE 4.3:

The sequence $X = (-1, 15, 21, 23)$ is a 4-cover of $Y = (-10, -5, -2, -1, 4, 5, 10, 12, 20, 22, 26, 31, 50)$. Consider, for example, the two consecutive elements -1 and 15 of $X_\infty = (-\infty, -1, 15, 21, 23, +\infty)$. The total number of elements of Y between -1 and 15 is 4. □

Lemma 4.3: *Let A and B be two sorted sequences of lengths n and m, respectively, and let X be a c-cover of A and B for some constant c. If $rank(X : A)$ and $rank(X : B)$ are known, then the problem of merging A and B can be solved in $O(1)$ time, using $O(|X|)$ operations.*

Proof: Let $X = (x_1, \ldots, x_s)$, $rank(X : A) = (r_1, r_2, \ldots, r_s)$ and $rank(X, B) = (t_1, t_2, \ldots, t_s)$. For each i, $1 \leq i \leq s + 1$, let $A_i = (a_{r_{i-1}+1}, \ldots, a_{r_i})$, and $B_i = (b_{t_{i-1}+1}, \ldots, b_{t_i})$, where $r_0 = t_0 = 0$, and $r_{s+1} = n$, $t_{s+1} = m$. Figure 4.4 illustrates the partitions of A and B. Note that A_i or B_i could be

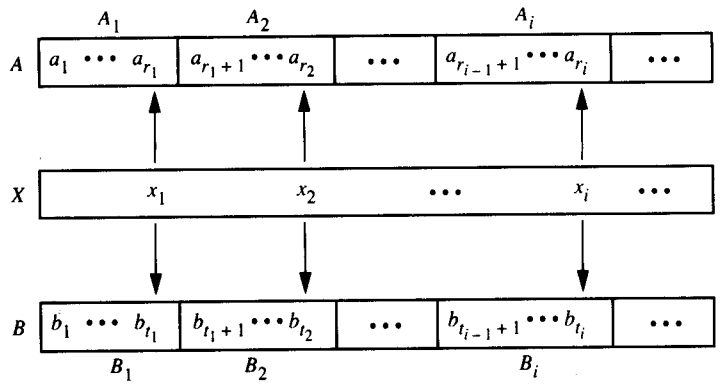


FIGURE 4.4
A merge of the two sorted sequences A and B with the cover X. The arrows indicate rank information as follows: $r_i = rank(x_i : A)$ and $t_i = rank(x_i : B)$. Since X is a c-cover, we have $|A_i|, |B_i| \leq c$.

empty (if $r_{i-1} = r_i$ or $t_{i-1} = t_i$, respectively). We now show how to compute $rank(A : B)$. Computing $rank(B : A)$ can be done in a similar fashion.

Suppose that $A_i \neq \emptyset$ and let $a \in A_i$. Then, $rank(a : B) = t_{i-1} + rank(a : B_i)$, since $b_{t_{i-1}} \leq x_{i-1} < a_{r_{i-1}+1} \leq a \leq a_{r_i} \leq x_i < b_{t_i+1}$. Hence, the problem reduces to determining $rank(a : B_i)$. But $|B_i| \leq c$, since X is a c-cover of B. Hence, a can be ranked in B_i in $O(1)$ sequential time. Therefore, the array $rank(A : B)$ can be found in $O(1)$ time using a linear number of operations. □

Remark 4.3: Suppose we know that X is a c-cover of B, and we are given $rank(A : X)$ and $rank(X : B)$. Then, the technique used in the proof of Lemma 4.3 allows us to determine $rank(A : B)$ in $O(1)$ time, using $O(|A| + |X|)$ operations. □

4.3 Sorting

The problem of **sorting** a sequence X is the process of rearranging the elements of X such that they appear in nondecreasing or nonincreasing order. The problem of sorting has been studied extensively in the literature because

of its many important applications and because of its intrinsic theoretical significance. Many solution strategies have been studied in some depth and their actual performances reported.

In this section, we restrict ourselves to the modest goal of providing two possible parallel implementations of the **merge-sort strategy** on the PRAM model, each requiring $O(n \log n)$ operations and hence optimal. The first implementation yields a simple parallel-sorting algorithm that runs in $O(\log n \log \log n)$ time. The second is significantly more involved and depends on an intricate pipelining scheme; however, it results in an $O(\log n)$ time sorting algorithm.

An additional parallel-sorting algorithm, called *bitonic sorting*, is described in Section 4.4. In Chapter 9 (Section 9.6), we describe and analyze a *randomized quicksort* algorithm.

4.3.1 A SIMPLE OPTIMAL SORTING ALGORITHM

As its name indicates, the *merge-sort strategy* is based on a merging procedure that is used to sort successively a number of larger and larger nonoverlapping subsequences until the whole sequence is sorted. One possible way to implement this strategy, referred to as **two-way merge sort,** is to start by sorting pairs of elements of the given sequence X, and then to sort every pair of consecutive pairs, and so on, until X is sorted.

The two-way merge-sort algorithm can be also viewed as an application of the divide-and-conquer strategy that consists of (1) dividing the input sequence X into two subsequences, X_1 and X_2, of approximately the same size; (2) sorting X_1 and X_2 separately; and finally (3) merging the two sorted sequences.

The sequence of operations required by the two-way merge-sort algorithm can be represented by a binary tree as follows. Let T be a balanced binary tree with n leaves. The elements of X are distributed among the leaves, one per leaf. The nodes at height 1 represent the lists we obtain by merging the pairs of consecutive elements contained in the children nodes (leaves, in this case). More generally, each internal node represents the subsequence we obtain by merging the subsequences generated at the children nodes. Hence, each internal node represents the sorted list of the elements stored in its subtree.

Since we are interested in a parallel implementation of the merge-sort algorithm, our problem can be rephrased as follows. For each node v of the balanced binary tree T, compute the sorted list $L[v]$ containing all the elements stored in the subtree rooted at v. Clearly, the root will contain the sorted list.

This process can be achieved in a fashion similar to that used in Algorithm 3.8 (the basic range-minima algorithm), which computes, for each node

v of a balanced binary tree, the prefix minima and the suffix minima of the elements contained in the subtree rooted at v. The only difference lies in the merging procedure. For our sorting algorithm, we use the optimal $O(\log \log n)$ time merging procedure described in Section 4.2.

The formal algorithm is given next. The node (h,j) of a binary tree is the jth node at height h ordered in a left to right fashion.

ALGORITHM 4.3
(Simple Merge Sort)

Input: *An array X of order n, where $n = 2^l$ for some integer l.*
Output: *A balanced binary tree with n leaves such that, for each $0 \leq h \leq \log n$, $L(h,j)$ contains the sorted subsequence consisting of the elements stored in the subtree rooted at node (h,j), for $1 \leq j \leq n/2^h$. That is, node (h,j) contains the sorted list of the elements $X(2^h(j-1)+1), X(2^h(j-1)+2), \ldots, X(2^h j)$.*

begin
1. **for** $1 \leq j \leq n$ **pardo**
   Set $L(0,j) := X(j)$
2. **for** $h = 1$ **to** $\log n$ **do**
   for $1 \leq j \leq n/2^h$ **pardo**
   Merge $L(h-1, 2j-1)$ *and* $L(h-1, 2j)$ *into the sorted list* $L(h,j)$

end

EXAMPLE 4.4:

Consider the sequence $X = (12, -5, -7, 51, 6, 28, 3, -8)$. Fig. 4.5 shows the binary tree with the initial contents of the leaves. During iteration $h = 1$, we get $L(1,1) = (-5, 12), L(1,2) = (-7, 51), L(1,3) = (6, 28)$ and $L(1,4) = (-8, 3)$. The next iteration causes the following two lists to be created: $L(2,1) = (-7, -5, 12, 51)$ and $L(2,2) = (-8, 3, 6, 28)$. Finally, the list at the root is generated and is given by $L(3,1) = (-8, -7, -5, 3, 6, 12, 28, 51)$. □

Theorem 4.3: *For each node v of the balanced tree T, Algorithm 4.3 generates the sorted list $L[v]$ consisting of the elements stored in the subtree rooted at v. The running time of the algorithm is $O(\log n \log \log n)$, and the total number of operations used is $O(n \log n)$. Hence, this algorithm is optimal.*

Proof: The correctness proof of Algorithm 4.3 is straightforward.

The number of iterations executed at step 2 is $O(\log n)$. Since the total number of elements involved at each level is n, each iteration takes $O(\log \log n)$ time, using a total of $O(n)$ operations if we use the optimal fast merging algorithm presented in Section 4.2. Hence, the theorem follows. □

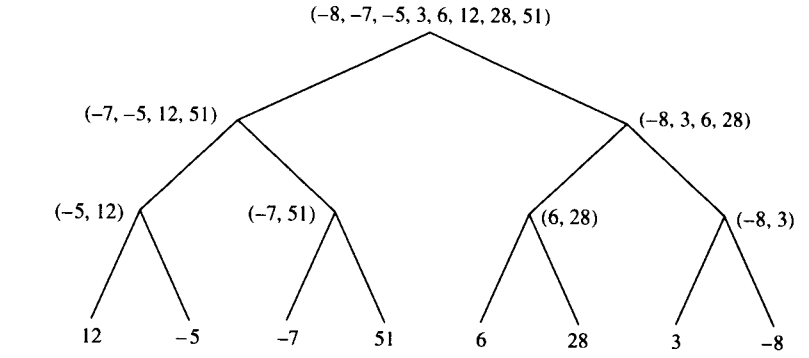


FIGURE 4.5
A merge-sort tree for eight elements. Each node contains the sorted list containing the elements stored in its subtree.

Remark 4.4: The space used by Algorithm 4.3 can be made $O(n)$ if we are interested in only the final sorted list. Once iteration h is completed, the nodes at height $h - 1$ will not be needed; hence, their space can be used to store the nodes at height h. □

Corollary 4.2: *Sorting a sequence of n elements can be done optimally in $O(\log n \log\log n)$ time.*

PRAM Model: The only nontrivial operation used in Algorithm 4.3 is the $O(\log\log n)$ time merging procedure of Section 4.2. Hence, this algorithm requires the CREW PRAM model. Had we used an EREW PRAM merging procedure, the corresponding sorting algorithm would have been an EREW PRAM algorithm. □

4.3.2 *AN OPTIMAL O(LOG n) TIME SORTING ALGORITHM

This section is devoted to the derivation of an $O(\log n)$ time optimal sorting algorithm. Actually, we address a slightly more general problem; its solution will provide us with an $O(\log n)$ time optimal sorting algorithm. In addition, several applications will make use of the general version.

The general formulation is as follows. Let T be a binary tree such that each leaf u contains an unsorted list $A(u)$ drawn from a linearly ordered set. We consider the problem of determining, for each internal node v, the sorted

list $L[v]$ that contains all the elements stored in the subtree rooted at v. Note that the initial list $A(u)$ of a leaf u could be empty.

EXAMPLE 4.5:

Consider the tree T shown in Fig. 4.6(a). The list that should be generated at the indicated node is given by $(-9, -7, -6, 2, 5)$. □

We start by making a couple of transformations. We replace each leaf u with a balanced binary tree with $|A(u)|$ leaves such that each element of $A(u)$ is stored in one of the leaves. The height of T has increased by $O(\log(\max_u |A(u)|))$, but each leaf of T now contains at most one element.

The second transformation is to force each internal node to have two children. If this is not the case, a leaf containing no elements can be inserted.

EXAMPLE 4.6:

Applying the two transformations to the tree T given in Fig. 4.6(a), we obtain the tree T' shown in Fig. 4.6(b). □

Therefore, we assume for the remainder of this section that T is a binary tree such that at most one element is stored in a leaf node and every internal node has exactly two children.

Pipelined Merge-Sort Algorithm. We introduced in Section 4.2 the notion of a c-cover, and the integer array $rank(A : B)$ corresponding to ranking a sorted list A in a sorted list B. Before proceeding, we need to make the following additional definition.

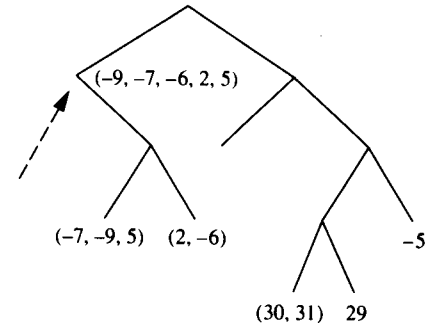
Given a sorted list L, the **c-sample** of L, denoted by $sample_c\ (L)$, is the sorted sublist of L consisting of every cth element of L; that is, if $L = (l_1, l_2, \ldots)$, then $sample_c\ (L) = (l_c, l_{2c}, \ldots)$.

EXAMPLE 4.7:

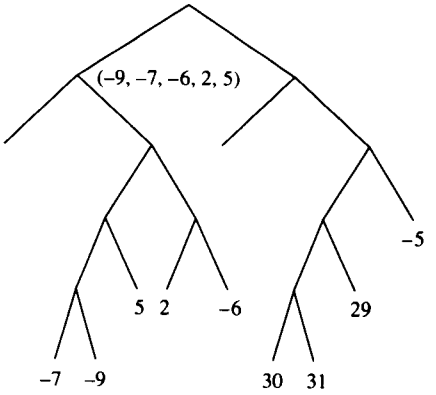
Let $L = (4, 7, 8, 9, 11, 15, 38)$. Then $sample_3\ (L)$ is given by $sample_3\ (L) = (8, 15)$. □

The parallel merge-sort strategy presented earlier is based on a forward traversal of the binary tree such that, for all vertices at a given height h, the lists $L[v]$ are *completely determined* before processing of the nodes at height $h + 1$ begins.

The **pipelined (or cascading) divide-and-conquer strategy** consists of determining $L[v]$ over a number of stages such that, at stage s, $L_s[v]$ is an approximation of $L[v]$ that will be improved at the next stage $s + 1$. At the



(a)



(b)

FIGURE 4.6
The tree for Examples 4.5 and 4.6. (a) An input tree for the general sorting problem; the arrow points to a node with its expected sorted list. (b) The tree T' after application of the two transformations: we replace each leaf containing a list by a balanced binary tree, and force each internal node to have exactly two children.

same time, a *sample* of $L_s[v]$ is propagated upward to be used for obtaining approximations of the lists to be generated at higher heights. The success of this method is due to the intricate combination of pipelining and the efficient merging of sample lists.

We now describe the procedure for determining $L_s[v]$ precisely; later, we provide a correctness proof and an analysis of the resources required.

Let $L_0[v] = \emptyset$ if v is an internal node; otherwise, $L_0[v]$ consists of the item (if any) stored at the leaf v. Let the **altitude** of a node v be defined as $alt(v) = h(T) - level(v)$, where $h(T)$ is the height of T, and, as usual, $level(v)$ is the length of the path from the root to v. The list stored at an internal node v will be updated over the stages s satisfying $alt(v) \leq s \leq 3alt(v)$.

We say that v is **active** during stage s if $alt(v) \leq s \leq 3alt(v)$. The algorithm will update the list $L_s[v]$ such that node v will be **full**—that is, $L_s[v] = L[v]$—when $s \geq 3alt(v)$. It is clear that, if this invariant can be maintained, then, after $3h(T)$ stages, the node at the root will be full, and all the nodes will contain their sorted lists.

An additional notation is needed before a description of the algorithm is given. Define $Sample(L_s[x])$ for an arbitrary node x as follows.

$$Sample(L_s[x]) = \begin{cases} sample_4\ (L_s[x])\ if\ s \leq 3alt(x); \\ sample_2\ (L_s[x])\ if\ s = 3alt(x) + 1; \\ sample_1\ (L_s[x])\ if\ s = 3alt(x) + 2. \end{cases}$$

Therefore, $Sample(L_s[x])$ is the sublist consisting of every fourth element of $L_s[x]$ until it becomes full; then $Sample(L_s[x])$ is every other element in the following stage (that is, stage $3alt(x) + 1$), and every element in stage $3alt(x) + 2$.

We next give a description of a general stage of the pipelined merge-sort algorithm that maintains the stated invariant. That is, for each node v, $L_s[v]$ will be full when $s \geq 3alt(v)$.

ALGORITHM 4.4

(Pipelined Merge Sort)

Input: *For each node v of a binary tree, a sorted list $L_s[v]$ such that v is full whenever $s \geq 3alt(v)$.*

Output: *For each node v, a sorted list $L_{s+1}[v]$ such that v is full whenever $s \geq 3alt(v) - 1$.*

Algorithm for stage $(s + 1)$:

begin

for *all active nodes v* **pardo**

1. Let u and w be the children of v. Set $L'_{s+1}[u] = Sample(L_s[u])$ and $L'_{s+1}[w] = Sample(L_s[w])$.

2. Merge the two lists $L'_{s+1}[u]$ and $L'_{s+1}[w]$ into the sorted list $L_{s+1}[v]$.

end

EXAMPLE 4.8:

Let T be the binary tree shown in Fig. 4.7(a). The lists corresponding to a set of selected stages are shown in the table of Fig. 4.7(b). Note that, initially, no changes occur until stage $s = 3$. At the end of stage $s = 3$, all the nodes of altitude 1 (nodes v_5 and v_6, in this case) become full. Consider, for example, node v_5. Since $alt(v_5) = 1$, v_5 is active during stage $s = 3$. In this case, $L_3'[v_1] = sample_1\ (L_2[v_1]) = (7)$, and similarly $L_3'[v_2] = (8)$. Hence, $L_3[v_5] = (7, 8)$. During this stage, we also obtain $L_3[v_6] = (1, 6)$.

The lists generated at several later stages are shown in the figure. Notice that, at the end of stage $s = 6$, the nodes at altitude 2 become full; at the end of stage $s = 9$, the nodes at altitude 3 become full. The root v_{21} is active for all stages s, $5 \le s \le 15$. However, $L_s[v_{21}]$ remains empty until stage $s = 13$ since, at each of the previous stages, the lists of the children nodes v_{19} and v_{20} contain less than four elements. At the end of stage $s = 12$, nodes v_{19} and v_{20} become full and each contain at least four elements. Hence, at stage $s = 13$, $L_s[v_{21}] = (5, 15)$, which results from the merging of $sample_4(L[v_{19}])$ and $sample_4(L[v_{20}])$. At the end of stage $s = 15$, $L_{15}[v_{21}]$ consists of the sorted list of all the items stored in the tree. □

We are ready to show that Algorithm 4.4 works correctly.

Lemma 4.4: *Let v be an arbitrary node of the binary tree T. Then, at the end of stage $s = 3alt(v)$ of Algorithm 4.4, v becomes full; that is, $L_s[v] = L[v]$.*

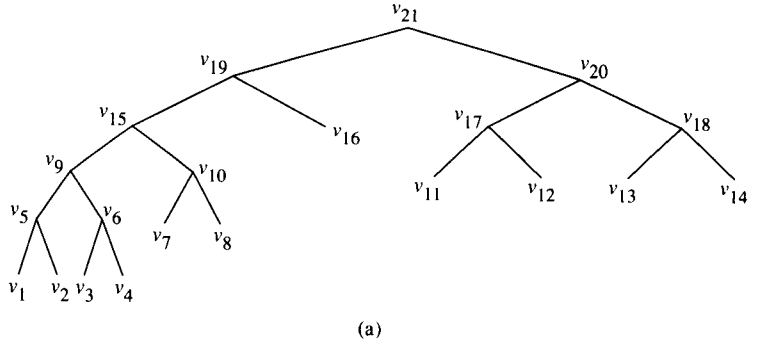


FIGURE 4.7
The tree for Example 4.8. (a) A binary tree.

v	$s = 0$	$s = 3$	$s = 5$	$s = 6$	$s = 8$	$s = 9$	$s = 11$	$s = 13$
1	(7)	(7)	(7)	(7)	(7)	(7)	(7)	(7)
2	(8)	(8)	(8)	(8)	(8)	(8)	(8)	(8)
3	(6)	(6)	(6)	(6)	(6)	(6)	(6)	(6)
4	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)
5	0	(7, 8)	(7, 8)	(7, 8)	(7, 8)	(7, 8)	(7, 8)	(7, 8)
6	0	(1, 6)	(1, 6)	(1, 6)	(1, 6)	(1, 6)	(1, 6)	(1, 6)
7	(5)	(5)	(5)	(5)	(5)	(5)	(5)	(5)
8	(3)	(3)	(3)	(3)	(3)	(3)	(3)	(3)
9	0	0	(6, 8)	(1, 6, 7, 8)	(1, 6, 7, 8)	(1, 6, 7, 8)	(1, 6, 7, 8)	(1, 6, 7, 8)
10	0	0	0	(3, 5)	(3, 5)	(3, 5)	(3, 5)	(3, 5)
11	(4)	(4)	(4)	(4)	(4)	(4)	(4)	(4)
12	(10)	(10)	(10)	(10)	(10)	(10)	(10)	(10)
13	(9)	(9)	(9)	(9)	(9)	(9)	(9)	(9)
14	(15)	(15)	(15)	(15)	(15)	(15)	(15)	(15)
15	0	0	0	0	(5, 6, 8)	(1, 3, 5, 6, 7, 8)	(1, 3, 5, 6, 7, 8)	(1, 3, 5, 6, 7, 8)
16	(2)	(2)	(2)	(2)	(2)	(2)	(2)	(2)
17	0	0	0	0	0	(4, 10)	(4, 10)	(4, 10)
18	0	0	0	0	0	(9, 15)	(9, 15)	(9, 15)
19	0	0	0	0	0	0	(3, 6, 8)	(1, 2, 3, 5, 6, 7, 8)
20	0	0	0	0	0	0	(10, 15)	(4, 9, 10, 15)
21	0	0	0	0	0	0	0	(5, 15)

(b)

FIGURE 4.7 (*continued*)
(b) The lists arising during the execution of the indicated stages of the pipelined merge-sort algorithm.

Proof: The proof is by induction on $alt(v)$. The claim is obviously true for all nodes satisfying $alt(v) = 0$, since they all are leaves satisfying initially $L_0[v] = L[v]$.

Let v be a node with $alt(v) = k > 0$. If v is a leaf, then $L_0[v] = L[v]$, and there is nothing to prove. Assume that v is an internal node with children u and w. Clearly, $alt(u) = alt(w) = k - 1$. By the induction hypothesis, u and w will become full at stage s' such that $s' = 3(k - 1)$. During stage $s' + 1$, $Sample(L_{s'}[u])$ and $Sample(L_{s'}[w])$ will be merged to form $L_{s'+1}[v]$. But $Sample(L_{s'}[u]) = sample_4\ (L_{s'}[u]) = sample_4\ (L[u])$, and similarly for w. During stage $s' + 2$, $Sample(L_{s'+1}[u])$ will consist of every other element of $L[u]$; during stage $s' + 3$, $Sample(L_{s'+2}[u]) = L[u]$. Similar arguments hold for node w. Hence, at stage $s' + 3 = 3k = 3alt(v)$, $L_{s'+3}[v] = L[v]$, and the lemma follows by induction. □

The following lemma states that the size of each list can grow at most by roughly a factor of 2 after each stage. The proof is a simple induction on s and is left to the reader in Exercise 4.22.

Lemma 4.5: *Let v be an arbitrary node of T and let $s \geq 1$. Then,* $|L_{s+1}[v]| \leq 2|L_s[v]| + 4$. □

Remark 4.5: For a given stage s, the total number of elements stored in all the active nodes of T is given by $n_s = \sum_{v\ active} |L_s[v]| = \sum_{\lfloor \frac{s}{3} \rfloor \leq alt(v) \leq s} |L_s[v]|$. Note that, if a node v is full, where $alt(v) = \lfloor s/3 \rfloor$, none of its ancestors can be full. Thus, $\sum_{alt(v) = \lfloor \frac{s}{3} \rfloor} |L_s[v]| \leq n$, where n is the number of leaves in T. Consider the active nodes at the level just above the level of these full nodes. There can be at most $n/2$ elements stored in these nodes, and at most $n/4$ elements stored at the level above them, and so on. Therefore, $n_s = O(n)$. □

Implementation Detail and Analysis. The only essential details left are to show how to perform step 2 of Algorithm 4.4 in $O(1)$ time, using $O(n_s) = O(n)$ operations.

We have already seen (Lemma 4.3) that the merging of two sorted lists A and B can be done optimally in $O(1)$ time if we are given a c-cover X for A and B and if $rank(X : A)$ and $rank(X : B)$ are also given as a part of the input. Guided by this observation, we next show that, for each node v, the list $L_s[v]$ is a 4-cover for $L'_{s+1}[u]$ and for $L'_{s+1}[w]$, where u and w are the children of v. We later show how the two arrays $rank(L_s[v] : L'_{s+1}[u])$ and $rank(L_s[v] : L'_{s+1}[w])$ can be generated efficiently.

Covers for the Lists to be Merged. We start by establishing the fact that $Sample(L_{s-1}[v]) = L'_s[v]$ is a 4-cover of $Sample(L_s[v]) = L'_{s+1}[v]$.

Lemma 4.6: *Let v be an arbitrary node of T and let $s \geq 1$. Then, $L'_s[v]$ is a 4-cover of $L'_{s+1}[v]$.*

Proof: We prove a slightly stronger result. Let $[a, b]$ be an interval with $a, b \in (-\infty, L'_s[v], +\infty)$. We say that $[a, b]$ **intersects** $(-\infty, L'_s[v], +\infty)$ in k items (or that there are k items **in common**) if the number of elements $x \in (-\infty, L'_s[v], +\infty)$ such that $a \leq x \leq b$ is equal to k. We use induction on s to establish the following claim.

Claim: If $[a, b]$ intersects $(-\infty, L'_s[v], +\infty)$ in $k \geq 2$ items, then $[a, b]$ intersects $L'_{s+1}[v]$ in at most $2k$ items.

Proof of the Claim: The base case $s = 1$ is trivial, since no list has more than one element, and hence both $L'_s[v]$ and $L'_{s+1}[v]$ are empty.

Assume that the induction hypothesis holds up to stage $s - 1$; that is, for any stage $t < s$, we know that any interval $[a', b']$ with $a', b' \in (-\infty, L'_t[v], +\infty)$ intersects $L'_{t+1}[v]$ in at most $2h$ items, where h is the number of items common between $[a', b']$ and $(-\infty, L'_t[v], +\infty)$. We show that the claim holds for stage s.

Let $[a, b]$ be an interval with $a, b \in (-\infty, L'_s[v], +\infty)$ such that the number of common items is k. Assume that $s \leq 3alt(v)$. The case where $s \geq 3alt(v) + 1$ is straightforward, since $L_{s-1}[v] = L_s[v] = L[v]$. Since $s \leq 3alt(v), L'_s[v] = sample_4(L_{s-1}[v])$, and hence $[a, b]$ intersects $(-\infty, L_{s-1}[v], +\infty)$ in $4k - 3$ items.

Recall that we obtained $L_{s-1}[v]$ by merging $L'_{s-1}[u]$ and $L'_{s-1}[w]$, where u and w are the children of v. Hence, the items belonging to $[a, b]$ and $L_{s-1}[v]$ must have come from the set $L'_{s-1}[u] \cup L'_{s-1}[w]$, where each list is viewed as a set of elements. Let $[a_1, b_1]$ be the smallest interval containing $[a, b]$ such that $a_1, b_1 \in (-\infty, L'_{s-1}[u], +\infty)$. Similarly define $[a_2, b_2]$ such that $a_2, b_2 \in (-\infty, L'_{s-1}[w], +\infty)$. Let p be the number of elements in common between $[a_1, b_1]$ and $(-\infty, L'_{s-1}[u], +\infty)$. Similarly, define q to be the number of items in common between $[a_2, b_2]$ and $(-\infty, L'_{s-1}[w], +\infty)$. Since we are assuming that all the elements are distinct, we obtain that $p + q \leq 4k - 1 (= (4k - 3) + 2$, since two additional elements from $\{a_1, b_1, a_2, b_2\}$ are included).

By the induction hypothesis, $[a_1, b_1]$ intersects $L'_s[u]$ in at most $2p$ elements, and $[a_2, b_2]$ intersects $L'_s[w]$ in at most $2q$ elements (see Fig. 4.8 for a pictorial representation of the relationships among the different lists involved). Now $L_s[v]$ is just the list obtained after merging of $L'_s[u]$ and $L'_s[w]$. Hence, $[a, b]$ intersects $L_s[v]$ in at most $2p + 2q \leq 8k - 2$ elements. Since $L'_{s+1}[v] = sample_4(L_s[v])$, we obtain that $[a, b]$ intersects $L'_{s+1}[v]$ in at most $2k$ items, and the claim follows.

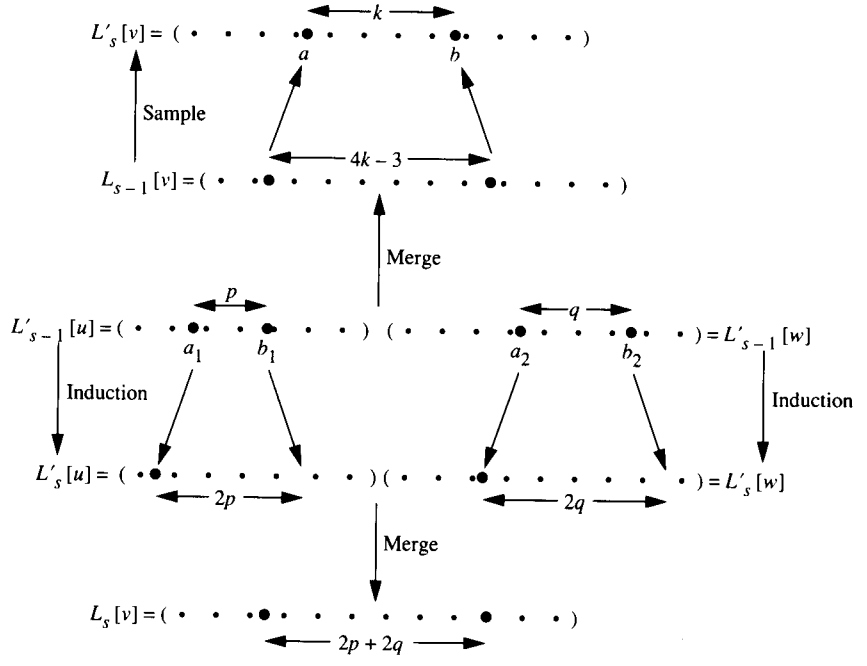


FIGURE 4.8
Illustration of the inductive proof of the fact that, if $[a, b]$ intersects $L'_s[v]$ in k elements, then $[a, b]$ intersects $L_s[v]$ in at most $2p + 2q \leq 8k - 2$ elements. This result implies that $[a, b]$ intersects $L'_{s+1}[v]$ in at most $2k$ items.

We now complete the proof of the lemma. Let $a, b \in (-\infty, L'_s[v], +\infty)$, such that a and b are adjacent. Hence, $[a, b]$ intersects $(-\infty, L'_s[v], +\infty)$ in exactly two items. As shown in the claim, $[a, b]$ intersects $L'_{s+1}[v]$ in at most four elements; hence, $L'_s[v]$ is a 4-cover of $L'_{s+1}[v]$. □

Corollary 4.3: *For each internal node v of T, and for each stage $s \geq alt(v)$, $L_s[v]$ is a 4-cover of $L'_{s+1}[u]$ and $L'_{s+1}[w]$, where u and w are the children of v.*

Proof: Let a and b be two adjacent elements of $(-\infty, L_s[v], +\infty)$. Recall that we obtain $L_s[v]$ by merging $L'_s[u]$ and $L'_s[w]$. Let $[a', b']$ be the smallest interval containing a and b such that $a', b' \in (-\infty, L'_s[u], +\infty)$. Clearly, a' and b' are adjacent in $(-\infty, L'_s[u], +\infty)$. Hence, by Lemma 4.6, there are at

most four elements in $L'_{s+1}[u]$ between a' and b'. This result implies that there are at most four elements in $L'_{s+1}[u]$ between a and b, which proves that $L_s[v]$ is a 4-cover for $L'_{s+1}[u]$.

We can establish in a similar fashion that $L_s[v]$ is a 4-cover of $L'_{s+1}[w]$. □

Efficient Merging of the Samples. The next lemma shows how to maintain efficiently certain rank information, which will be used to perform the merging operations fast.

Lemma 4.7: *Let $s \geq 2$ be a given stage number of Algorithm 4.4. Suppose that, for every internal node v of T and its two children u and w, we are given the following information:*

1. $rank(L'_s[v] : L'_{s+1}[v])$
2. $rank(L'_s[u] : L'_s[w])$
3. $rank(L'_s[w] : L'_s[u])$

Then, using $O(|L'_{s+1}[u]| + |L'_{s+1}[w]|)$ operations, we can compute the following information in $O(1)$ time:

1. $rank(L'_{s+1}[v] : L'_{s+2}[v])$
2. $rank(L'_{s+1}[u] : L'_{s+1}[w])$
3. $rank(L'_{s+1}[w] : L'_{s+1}[u])$

Proof: We first show how to obtain $rank(L'_{s+1}[u] : L'_{s+1}[w])$ within the stated bounds. Actually, the proof follows from Remark 4.3; for clarity, however, we provide the proof here.

Consider the array $rank(L'_s[u] : L'_{s+1}[u])$, which is given as a part of the input whenever u is an internal node. Since $L'_s[u]$ is a 4-cover of $L'_{s+1}[u]$, by Lemma 4.6, the number of elements in $L'_{s+1}[u]$ between any two consecutive elements of $(-\infty, L'_s[u], +\infty)$ is at most 4. Let S_i be the segment of $L'_{s+1}[u]$ consisting of all the elements between the ith and the $(i+1)$st elements—say, p and q—of $L'_s[u]$. Clearly, $|S_i| \leq 4$. We also know the ranks of p and q in $L'_s[w]$, since the array $rank(L'_s[u] : L'_s[w])$ is given as a part of the input. Let us denote $rank(p : L'_s[w])$ by s_1, and $rank(q : L'_s[w])$ by s_2. Therefore, the elements of S_i are known to lie in the segment S'_i of $L'_s[w]$ determined by the indices s_1 and s_2 (see Fig. 4.9 for an illustration).

The problem of generating $rank(L'_{s+1}[u] : L'_s[w])$ reduces to determining the relative ranks of each element of S_i in the corresponding block S'_i. Since $|S_i| \leq 4$, however, we can use the parallel-search procedure (Algorithm 4.1) to determine the ranks of all the elements of S_i in S'_i. Such computation takes $O(1)$ time, using $O(|S'_i|)$ operations. Since, for $i \neq j$, S'_i and S'_j do not overlap, the array $rank(L'_{s+1}[u] : L'_s[w])$ can be obtained in $O(1)$ time, using $O(|L'_{s+1}[u]| + |L'_s[w]|)$ operations.

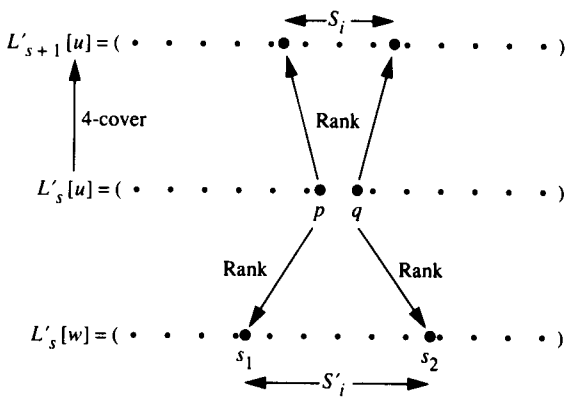


FIGURE 4.9
Illustration of how to compute $rank(L'_{s+1}[u] : L'_s[w])$. Note that $L'_s[u]$ is a 4-cover of $L'_{s+1}[u]$, and that the array $rank(L'_s[u] : L'_s[w])$ is given as part of the input. The elements p and q are the ith and the $(i + 1)$st elements of $L'_s[u]$.

We can easily determine the array $rank(L'_{s+1}[u] : L'_{s+1}[w])$ by using $rank(L'_s[w] : L'_{s+1}[w])$ and the fact that $L'_s[w]$ is a 4-cover of $L'_{s+1}[w]$. We can obtain the array $rank(L'_{s+1}[w] : L'_{s+1}[u])$ in a similar fashion.

We now consider the problem of determining $rank(L'_{s+1}[v] : L'_{s+2}[v])$. Recall that $L'_{s+1}[v]$ is just a sample of $L_s[v]$, and that $L_s[v]$ is obtained by merging of $L'_s[u]$ and $L'_s[w]$.

Let p be any element of $L'_s[u]$. We know the rank of p in $L'_s[u]$, of course. We also know the rank of p in $L'_s[w]$, since we are given the array $rank(L'_s[u] : L'_s[w])$. Since $rank(p : L_s[v]) = rank(p : L'_s[u]) + rank(p : L'_s[w])$, we know, for each $p \in L'_s[u]$, the latter's location in the array $L_s[v]$. Moreover, since the array $rank(L'_s[u] : L'_{s+1}[u])$ is also given, we know $rank(p : L'_{s+1}[u])$.

We can also obtain $rank(p : L'_{s+1}[w])$ as follows. Suppose that $rank(p : L'_s[w]) = r_1$. Then, p is between elements e and f at positions r_1 and $r_1 + 1$, respectively, of $L'_s[w]$. Since $rank(L'_s[w] : L'_{s+1}[w])$ is known, we can determine in $O(1)$ sequential time the boundaries of the segment S_p of $L'_{s+1}[w]$ of all elements between e and f (see Fig. 4.10 for an illustration). Using the fact that $L'_s[w]$ is a 4-cover of $L'_{s+1}[w]$, we conclude that $|S_p| \leq 4$. Therefore, we can determine in $O(1)$ sequential time $rank(p : L'_{s+1}[w])$.

Since $rank(p : L_{s+1}[v]) = rank(p : L'_{s+1}[u]) + rank(p : L'_{s+1}[w])$, the rank of p in $L_{s+1}[v]$ can be determined in $O(1)$ sequential time. However, since $L'_{s+2}[v]$ is a sample of $L_{s+1}[v]$, we can obtain the rank of p in $L'_{s+2}[v]$ in $O(1)$ sequential time.

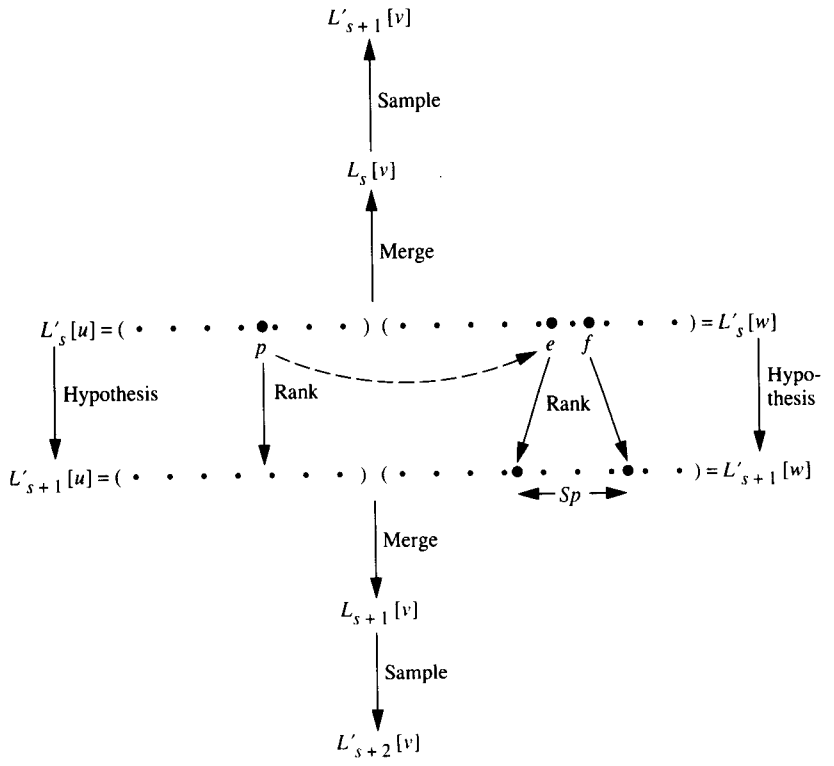


FIGURE 4.10
Determination of $rank(p : L'_{s+2}[v])$ in three steps: (1) determine $rank(p : L'_{s+1}[w])$ using $rank(p : L'_s[w])$, which identifies e and f, followed by ranking p in S_p; (2) determine $rank(p : L_{s+1}[v])$; and (3) use the fact that $L'_{s+2}[v]$ is a sample of $L_{s+1}[v]$ to deduce $rank(p : L'_{s+2}[v])$.

We can perform the same procedure for all the elements in $L'_s[w]$. Therefore, we can determine $rank(L_s[v] : L'_{s+2}[v])$ in $O(1)$ time, using a linear number of operations. Once this value is determined, generating the array $rank(L'_{s+1}[v] : L'_{s+2}[v])$ can be done easily in $O(1)$ time, using $O(|L'_{s+1}[v]|)$ operations. □

Corollary 4.4: *Under the hypothesis of Lemma 4.7, we can determine* $rank(L_s[v] : L'_{s+1}[u])$ *and* $rank(L_s[v] : L'_{s+1}[w])$ *in* $O(1)$ *time using a linear number of operations.*

Proof: For each $p \in L'_s[u]$, we know $rank(p : L'_s[w])$ and $rank(p : L'_{s+1}[u])$ from the given input (as listed in Lemma 4.7). Hence, we can trivially obtain $rank(p : L_s[v])$ and $rank(p : L'_{s+1}[u])$ for each $p \in L'_s[u]$.

On the other hand, for each $q \in L'_s[w]$, we can deduce $rank(q : L_s[v])$ and $rank(q : L'_{s+1}[u])$ in $O(1)$ sequential time by using $rank(L'_s[w] : L'_s[u])$, $rank(L'_s[u] : L'_{s+1}[u])$, and the fact that $L'_s[u]$ is a 4-cover of $L'_{s+1}[u]$. Hence, we can determine in $O(1)$ sequential time the rank of each element of $L_s[v]$ in $L'_{s+1}[u]$. We can in a similar fashion determine $rank(L_s[v] : L'_{s+1}[w])$. □

The algorithm to compute $L_{s+1}[v]$, for an arbitrary active node v with children u and w, can be summarized as follows.

As a part of the input to stage $s + 1$, we are given the following rank information: (1) $rank(L'_s[u] : L'_{s+1}[u])$, (2) $rank(L'_s[w] : L'_{s+1}[w])$, (3) $rank(L'_s[u] : L'_s[w])$, and (4) $rank(L'_s[w] : L'_s[u])$. The main steps are these:

1. Compute $rank(L_s[v] : L'_{s+1}[u])$ and $rank(L_s[v] : L'_{s+1}[w])$ (Corollary 4.4).
2. Merge $L'_{s+1}[u]$ and $L'_{s+1}[w]$ using $L_s[v]$ as a 4-cover for both lists (Lemma 4.3). The resulting list is $L_{s+1}[v]$.
3. Update the necessary input information for stage $s + 2$ (Lemma 4.7).

Putting the Pieces Together. Combining all the facts shown in this section, we obtain the following theorem.

Theorem 4.4: *Let T be a given binary tree such that each leaf v contains a list $A(v)$. Let $h(T)$ be the height of T, and let $m = \max_v |A(v)|$. Then, the pipelined merge-sort algorithm (Algorithm 4.4) generates, for each node v of T, a sorted list $L[v]$ containing all the items stored in the subtree rooted at v. The overall algorithm runs in $O(h(T) + \log m)$ time, using a total of $O((n_1 + n_2)(h(T) + \log m))$ operations, where n_1 is the number of nodes in T and n_2 is the total number of items in T.*

Proof: The tree T' that we obtain from T by replacing each list $A(v)$ with a balanced binary tree with $|A(v)|$ leaves is of height less than or equal to $h(T) + \log m$ and has no more than $n_1 + n_2$ leaves.

The pipelined merge-sort algorithm applied to T' consists of at most $3(h(T) + \log m)$ stages; each stage can be performed in $O(1)$ time, using $O(n_1 + n_2)$ operations (recall Remark 4.5, which states that the number of elements in all the active nodes of any stage is asymptotically the same as the number of leaves). Therefore, the running time is $O(h(T) + \log m)$, and the number of operations is $O((n_1 + n_2)(h(T) + \log m))$. □

Corollary 4.5: *Sorting n elements can be done in $O(\log n)$ time, using a total of $O(n \log n)$ operations.* □

PRAM Model: Concurrent read is used in the procedures outlined in Lemma 4.7. Hence, Algorithm 4.4 runs on the CREW PRAM model. This algorithm can be modified to run on the EREW PRAM, but the details required are nontrivial. □

Remark 4.6: It is easy to check that, after an arbitrary stage s of the pipelined merge-sort algorithm, we can obtain $rank(L_s[u] : L_s[v])$, for a node v and a child u, without any asymptotic increase in the resources used. Therefore, we can assume that, at the end of the algorithm, the array $rank(L[u] : L[v])$ is available. In addition, the lists generated at any sibling pair are cross-ranked. These facts will be used in Chapter 6 to derive optimal algorithms for several problems in computational geometry. □

4.4 Sorting Networks

The importance of sorting has stimulated a considerable amount of research in the development of special-purpose hardware for sorting. The *comparator-network* model for handling comparison problems is especially well suited for hardware implementation and has a rich theory. In this section, we shall introduce the classical **bitonic sorting network,** and outline several of its properties.

A **comparator network** is made up of **comparators,** where a comparator is a module whose two inputs are x and y and whose two ordered outputs are $\min\{x, y\}$ and $\max\{x, y\}$. A possible representation of a comparator is shown in Fig. 4.11(a). Note that the direction of the arrow used in this representation is significant.

An example of a comparator network that sorts is shown in Fig. 4.11(b). In this example, the input items are x_1, x_2, x_3, x_4, and the output items y_1, y_2, y_3, y_4 correspond to the sorted sequence.

The **size** of a comparator network is the number of comparators used in the network; the **depth** is the maximum number of comparators encountered when going from an input to an output. The sorting network shown in Fig. 4.11(b) is of size 5 and depth 3.

Our main goal in this section is to design comparator networks that sort in small depth and that have a small size. Given a comparator network that sorts, we can easily derive a parallel algorithm whose running time is asymptotically the same as the depth of the network, and whose total number of operations is asymptotically the same as the size of the network.

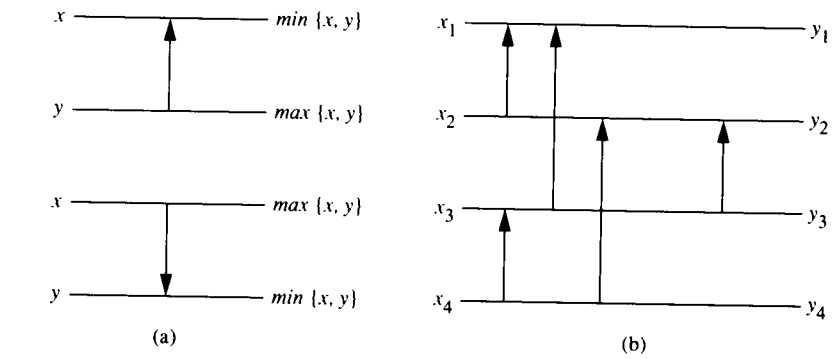


FIGURE 4.11
Comparator networks. (a) Single comparator. (b) A sorting network.

4.4.1 OBLIVIOUS ALGORITHMS

Closely related to the comparator network model is the class of **oblivious** comparison algorithms.

Suppose that the input is given in an array $A = (a_0, a_1, \ldots, a_{n-1})$. The control flow of an oblivious algorithm *does not depend on the particular values* of the a_i's. Hence, assume that the input is stored in locations 1, 2, ..., n. Each step of the algorithm consists of a set of comparisons between the pairs of values stored in locations $(i_1, j_1), (i_2, j_2), \ldots, (i_p, j_p)$, where all the indices are distinct. For each such pair, the minimum value is stored in the first location and the maximum is stored in the second location (or vice versa) as a result of the comparison. In particular, the locations considered at a given step do not depend on the outcome of the previous comparisons made.

It is straightforward to map oblivious sorting algorithms directly onto comparator networks.

4.4.2 BITONIC SEQUENCES

Let $X = (x_0, x_1, \ldots, x_{n-1})$ be a sequence of elements drawn from a linearly ordered set. Assume that $n = 2^k$, for some positive integer k. The sequence X is called **bitonic** if, for some positive integer $j < n$, we have $x_{j \bmod n} \leq x_{(j+1) \bmod n} \leq \cdots \leq x_{l \bmod n}$ and $x_{(l+1) \bmod n} \geq \cdots \geq x_{(j+n-1) \bmod n}$, for some l. That is, if we place the x_i's on a circle in the order they appear in X, the circle can be partitioned into two portions such that the subsequence defined by each portion is monotonic.

EXAMPLE 4.9:

Consider the sequence of numbers $X = (-5, -9, -10, -5, 2, 7, 35, 37)$. This sequence is bitonic, since $-10 < -5 < 2 < 7 < 35$ and $37 > -5 > -9$ ($j = 2$ and $l = 6$ in the previous definition). □

Unique Cross-Over Property of Bitonic Sequences. We now describe an important property of bitonic sequences, starting with special bitonic sequences.

Let $X = (x_0, x_1, \ldots, x_{n-1})$ be a sequence such that $x_0 \leq x_1 \leq \ldots \leq x_{\frac{n}{2}-1}$ and $x_{\frac{n}{2}} \geq x_{\frac{n}{2}+1} \geq \ldots \geq x_{n-1}$. This sequence can be represented by two vertical arrows, as shown in Fig. 4.12. The downward arrow represents a nondecreasing sequence, whereas the upward arrow represents a nonincreasing sequence.

There exists a unique horizontal line H that cuts the region between the two vertical arrows into two subregions, one labeled $\leq$ and the other labeled $>$, such that (1) an arbitrary pair of elements in the same subregion, one from the left column and the other from the right column, satisfies the relation indicated in its subregion; (2) every element in the left column of the subregion $\leq$ is less than or equal to every element in the left column of the subregion $>$. Property (2) holds also for the right column, as indicated by the inequalities. A horizontal line that induces two regions with these two properties is said to have the **unique cross-over property.**

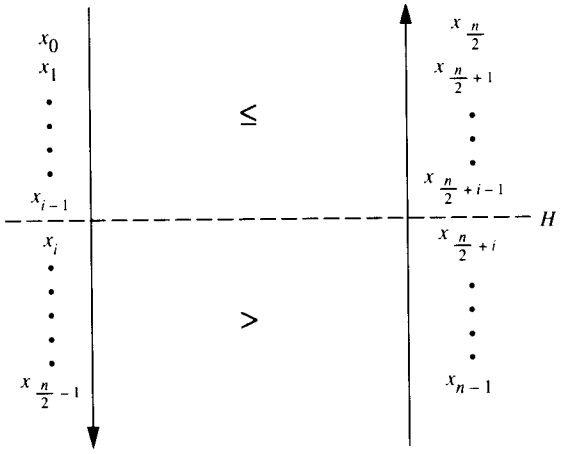


FIGURE 4.12
Representation of a bitonic sequence. The horizontal line H corresponds to a cut with the unique cross-over property.

To check these two properties, we observe the following. Suppose that there exists an index i such that $x_i > x_{\frac{n}{2}+i}$, where $0 \leq i \leq (n/2) - 1$. Let i be the smallest such index. Then, consider the horizontal cut between x_{i-1} and x_i (and also between $x_{\frac{n}{2}+i-1}$ and $x_{\frac{n}{2}+i}$). For $0 \leq j \leq i-1, x_j \leq x_{i-1} \leq x_{\frac{n}{2}+i-1} \leq x_{\frac{n}{2}+i-2} \leq \cdots \leq x_{\frac{n}{2}}$, and hence x_j is less than or equal to each element of the right-hand side in the subregion $\leq$. Note also that x_j is less than or equal to each element of the left side in subregion $>$. Similarly, we can verify that, for each j in the range $\frac{n}{2} \leq j \leq \frac{n}{2} + i - 1, x_j$ is greater than or equal to x_k for any k, where $\frac{n}{2} + i \leq k \leq n - 1$.

Notice that, when $i = 0$, the horizontal cut H is at the very top. If no index i exists such that $x_i > x_{\frac{n}{2}}$, then the horizontal cut is at the very bottom.

EXAMPLE 4.10:

Consider the bitonic sequence $X = (21, 18, 14, 10, -6, -4, 0, 1, 2, 19, 31, 30, 29, 22, 21, 21)$. When X is viewed as a circular list, we have the following two monotonic subsequences: $(-6, -4, 0, 1, 2, 19, 31)$ and $(30, 29, 22, 21, 21, 21, 18, 14, 10)$. A representation of X is shown in Fig. 4.13. The sequence X consists of a nonincreasing subsequence, followed by a nondecreasing subsequence, followed by a nonincreasing subsequence. The dotted line shown in the figure satisfies the unique cross-over property, as can be verified easily. □

We now show that the unique cross-over property holds in general.

Lemma 4.8: *Any bitonic sequence satisfies the unique cross-over property.*

Proof: Using the representation developed previously, we have to show that there always exists a horizontal cut that splits the region into two subregions $\leq$ and $\geq$, satisfying the following two properties:

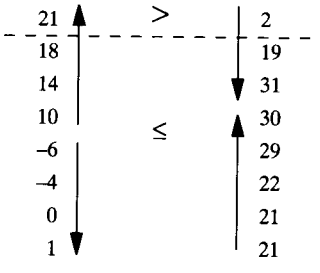


FIGURE 4.13
The horizontal cut for the given bitonic sequence.

1. Given any x_i and x_j in the same subregion, where x_i comes from the left side and x_j comes from the right side, then $x_i \leq x_j$ if the subregion is $\leq$; otherwise, $x_i \geq x_j$.
2. Given any x_i and x_j in the same column c—say, x_i in the subregion $\leq$ and x_j in the subregion $\geq$—then $x_i \leq x_j$ if c is the left column; otherwise, $x_i \geq x_j$.

Let $X = (x_0, x_1, \cdots, x_{n-1})$ be an arbitrary bitonic sequence. Without loss of generality, we assume that $x_0 \leq x_1 \leq \cdots \leq x_l$ and $x_{l+1} \geq x_{l+2} \geq \cdots \geq x_{n-1}$, for some index l. Otherwise, we can make a circular shift to put the given sequence in this form; the new sequence has a horizontal cut with the unique cross-over property if and only if the original sequence has the same property.

Assume that $l \geq n/2$; the other case can be dealt with similarly. The representation of the corresponding sequence is shown in Fig. 4.14.

We start by drawing a horizontal line H temporarily just below the two elements x_l and $x_{l-\frac{n}{2}}$. Clearly, the region above H must be labeled $\leq$, since $x_0 \leq x_1 \leq \cdots \leq x_{n/2} \leq \cdots \leq x_l$. We move H down until either (1) we find an index i such that $x_i > x_{\frac{n}{2}+i}$, where $0 \leq i \leq \frac{n}{2} - 1$, or (2) we determine that no such index exists and hence we have $x_{\frac{n}{2}-1} \leq x_{n-1}$. In case 1, we leave H just above x_i and $x_{\frac{n}{2}+i}$; in case 2, H is at the very bottom, below $x_{\frac{n}{2}-1}$ and x_{n-1}.

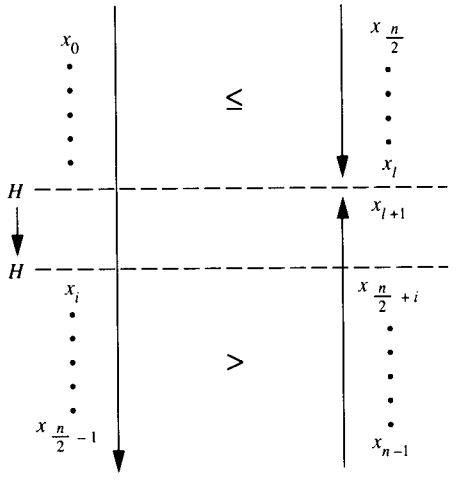


FIGURE 4.14
Construction of the horizontal cut of a bitonic sequence. The horizontal line H is moved down either to the very bottom or to just above x_i and $x_{\frac{n}{2}+i}$ such that $x_i > x_{\frac{n}{2}+i}$.

The reader can check that, in each case, H satisfies the desired unique cross-over property. □

Sorting of Bitonic Sequences. Given a bitonic sequence $X = (x_0, x_1, \cdots, x_{n-1})$, we define the following two operators on X:

$$L(X) = (\min\{x_0, x_{\frac{n}{2}}\}, \min\{x_1, x_{\frac{n}{2}+1}\}, \cdots, \min\{x_{\frac{n}{2}-1}, x_{n-1}\}),$$

$$R(X) = (\max\{x_0, x_{\frac{n}{2}}\}, \max\{x_1, x_{\frac{n}{2}+1}\}, \cdots, \max\{x_{\frac{n}{2}-1}, x_{n-1}\})$$

EXAMPLE 4.11:

Let $X = (21, 18, 14, 10, -6, -4, 0, 1, 2, 19, 31, 30, 29, 22, 21, 21)$ be the bitonic sequence of Example 4.10. Then, it is easy to check that $L(X) = (2, 18, 14, 10, -6, -4, 0, 1)$ and $R(X) = (21, 19, 31, 30, 29, 22, 21, 21)$. Note that each of $L(X)$ and $R(X)$ consists of a segment in the subregion $\leq$ and the complement segment in the subregion $>$ (see Fig. 4.13). □

Lemma 4.9: *Let X be a bitonic sequence. Then, both $L(X)$ and $R(X)$ are bitonic, and each element of $L(X)$ is smaller than each element of $R(X)$.*

Proof: By the unique cross-over property, there exists a horizontal cut—say, between x_i and x_{i+1} (and, of course, between $x_{\frac{n}{2}+i}$ and $x_{\frac{n}{2}+i+1}$)—that induces the two subregions $\leq$ and $\geq$. We shall assume that $\leq$ is on the top.

Given the properties of the horizontal cut, it is clear that $L(X) = (x_0, x_1, \cdots, x_i, x_{\frac{n}{2}+i+1}, \cdots, x_{n-1})$ and $R(X) = (x_{\frac{n}{2}}, x_{\frac{n}{2}+1}, \cdots, x_{\frac{n}{2}+i}, x_{i+1}, \cdots, x_{\frac{n}{2}-1})$. The fact that each element of $L(X)$ is no greater than each element of $R(X)$ follows immediately from the unique cross-over property. Since each of $L(X)$ and $R(X)$ is a subsequence of the bitonic sequence X, clearly $L(X)$ and $R(X)$ are bitonic. □

Based on Lemma 4.9, we deduce the following sorting algorithm, which can be viewed as an instance of the partitioning strategy.

ALGORITHM 4.5
(Bitonic Merge)

Input: *A bitonic sequence $X = (x_0, x_1, \cdots, x_{n-1})$ such that $n = 2^k$ for some integer k.*
Output: *The sequence X in sorted order.*
begin

1. **for** $0 \leq i \leq \frac{n}{2} - 1$ **pardo**
   Set l_i: $= min(x_i, x_{i+\frac{n}{2}})$
   Set r_i: $= max(x_i, x_{i+\frac{n}{2}})$
2. *Apply the algorithm recursively to each of the sequences $L(X) = (l_0, l_1, \cdots, l_{\frac{n}{2}-1})$ and $R(X) = (r_0, r_1, \cdots, r_{\frac{n}{2}-1})$.*

3. Output the sorted sequence $L(X)$ followed by the sorted sequence $R(X)$.

end

Bitonic Sorting Networks. Using Algorithm 4.5, we can construct the bitonic sorting network $B(n)$ to sort a bitonic sequence of length $n = 2^k$, for some integer k.

If $n = 2$, then $B(2)$ consists of just one comparator; otherwise, $B(n)$ consists of a column of $\frac{n}{2}$ comparators, followed by the two sorting networks $B\left(\frac{n}{2}\right)$, as shown in Fig. 4.15. The purpose of the column of $n/2$ comparators is to generate $L(X)$ on the first $n/2$ horizontal lines and $R(X)$ on the remaining horizontal lines. We are ready for the following theorem.

Theorem 4.5: *The bitonic sorting network $B(n)$ correctly sorts a bitonic sequence of length $n = 2^k$, where k is a positive integer. The depth of the network is $D(n) = \log n$, and the number of the comparators is $C(n) = \frac{n \log n}{2}$.*

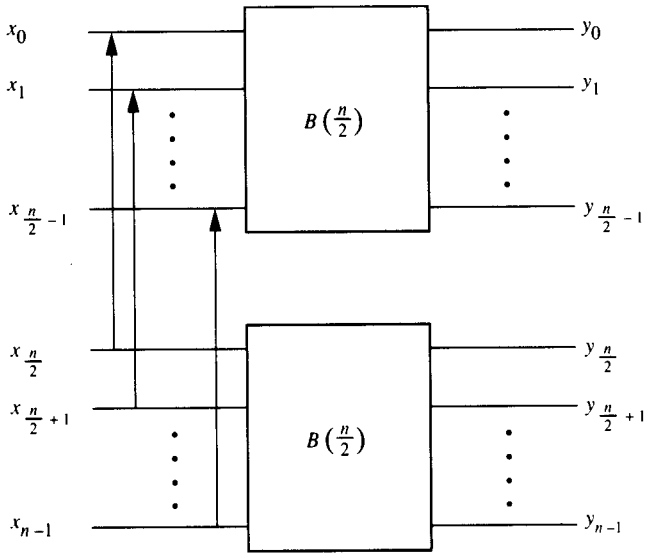


FIGURE 4.15
Construction of the bitonic sorting network. The minima of the corresponding pairs (that is, the set $L(X)$ defined in the text) appear as inputs to the top $B(n/2)$, and the maxima of the pairs (that is, the set $R(X)$) appear as inputs to the bottom $B(n/2)$.

Proof: Lemma 4.9 implies that the bitonic sorting network sorts correctly. The depth $D(n)$ and the size $C(n)$ satisfy the following recurrences:

$$\begin{cases} D(n) = 1 + D\left(\frac{n}{2}\right); \\ D(2) = 1. \end{cases}$$

$$\begin{cases} C(n) = \frac{n}{2} + 2C\left(\frac{n}{2}\right); \\ C(2) = 1. \end{cases}$$

It is easy to verify that $D(n) = \log n$ and $C(n) = \frac{n \log n}{2}$ satisfy these recurrence relations. □

Note that the bitonic sorting network sorts bitonic sequences, and *does not necessarily sort arbitrary sequences*. Hence, it is really more like a merging network than a sorting network. But once we construct a merging network, we can use it to build a sorting network by using the merge-sort method.

Given an arbitrary sequence $X = (x_0, x_1, \ldots, x_{n-1})$, start by using $\frac{n}{2}$ $B(2)$ networks in parallel on pairs of elements of X. Note that successive pairs should be sorted in opposite orders, so that the combined sequence of two consecutive pairs is bitonic. Next, sort each segment of four elements by using $\frac{n}{4}$ $B(4)$ networks in parallel, and so on. The bitonic sorting network to sort an arbitrary sequence of eight elements is shown in Fig. 4.16.

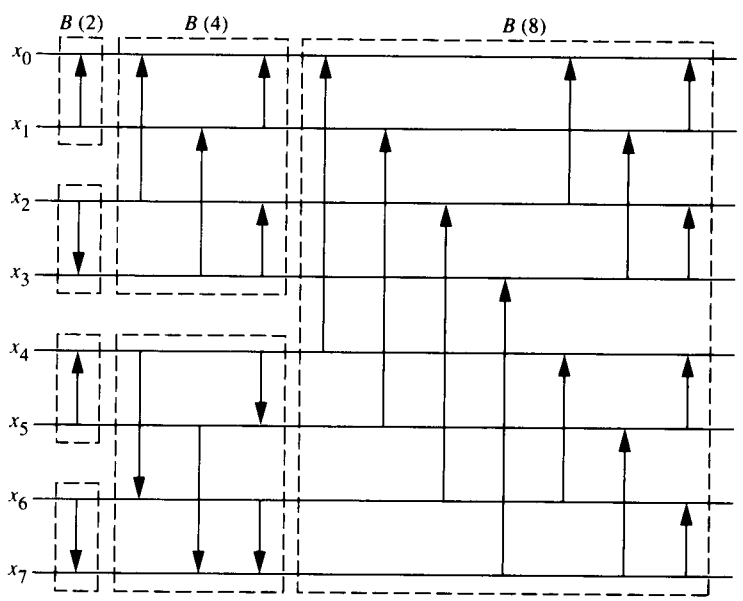


FIGURE 4.16
An eight-element bitonic sorting network. The network consists of a column of four $B(2)$'s, followed by two $B(4)$'s, followed by a single $B(8)$.

The depth of the bitonic sorting network to sort an arbitrary sequence of length $n = 2^k$ is $1 + 2 + 3 + \cdots + \log n = O(\log^2 n)$, and the number of comparators is given by $\sum_{i=1}^{k-1} \frac{n}{2^i} C(2^i) = O(n \log^2 n)$.

Therefore, we have the following theorem.

Theorem 4.6: *We can construct a network to sort an arbitrary sequence of length n in $O(\log^2 n)$ depth using $O(n \log^2 n)$ comparators.* □

The construction of another important sorting network based on the **odd-even merge algorithm** is outlined in Exercise 4.33.

The existence of faster and more economical (in hardware cost) networks is well established in the literature. The best known example is the AKS sorting network that sorts in $O(\log n)$ depth and has $O(n \log n)$ size. The derivation of this network is quite involved and is beyond the scope of this book.

4.5 Selection

Given a set of elements $A = (a_1, a_2, \ldots, a_n)$ and an integer k, where $1 \leq k \leq n$, the **selection problem** is to determine an element a_i of A such that $rank(a_i : A) = k$.

We have already examined the special cases $k = 1$ and $k = n$, corresponding, respectively, to computing the minimum and the maximum elements of A (Section 2.6). Another important special case is the determination of the **median,** corresponding to $k = \lceil \frac{n}{2} \rceil$. The selection problem is known to admit a linear-time sequential algorithm based on the divide-and-conquer strategy.

Recall that the problem of computing the maximum (or the minimum) element can be solved optimally in $O(\log \log n)$ time on the CRCW PRAM and in $O(\log n)$ time on the EREW PRAM (Section 2.6). However, finding the median (and hence solving the general selection problem) turns out to be more difficult. We shall show in Chapter 10 that the problem of determining the median requires $\Omega(\log n / \log \log n)$ time on the CRCW PRAM, using any polynomial number of processors.

In the remainder of this section, we shall present a parallel algorithm for the general selection problem that runs in $O(\log n \log \log n)$ time, using a linear number of operations. This algorithm can be viewed as the parallel version of a known optimal sequential algorithm.

Our selection problem can be solved easily if we preprocess our set by sorting its elements, and then output the kth element in the sorted list. If we use

the pipelined merge-sort algorithm, this approach leads to a fast $O(\log n)$ time algorithm; however, it uses a total of $O(n \log n)$ operations, which is nonoptimal.

To develop an optimal parallel algorithm, we use the *accelerated-cascading technique* (Section 2.6). Hence an optimal algorithm that reduces the size n of the input array to $O(n/\log n)$ is needed. Such an algorithm followed by the pipelined merge-sort algorithm will yield the desired optimal algorithm. The reader should note, in particular, the similarity between the solution strategies of the list ranking (Section 3.1) and the selection problems.

The idea behind the size-reducing algorithm is simple. Suppose an element a of A is identified with the property $\frac{n}{4} \leq rank(a : A) \leq \frac{3n}{4}$. Then, a induces a partition of A into three subsets A_1, A_2 and A_3 consisting of the elements smaller than, equal to, and larger than a, respectively. Let $s_i = |A_i|$, *for* $1 \leq i \leq 3$. Note that $s_1, s_3 \leq \frac{3n}{4}$, because $\frac{n}{4} \leq rank(a : A) \leq \frac{3n}{4}$. Based on the relative sizes of s_1, s_2, s_3, and k, we can now isolate the kth smallest element in one of the subsets A_1, A_2, or A_3. If the kth smallest element is in A_2, then we are done; otherwise, the size of the corresponding array has been reduced by at least a factor of 3/4.

Repeating this process $O(\log \log n)$ times reduces the size of A to $O(n/\log n)$. We can now use the pipelined merge-sort algorithm to identify the desired element.

The only essential detail left is the description of the method used to determine the element a. We can determine it by using *the median of the medians rule* as explained in steps 2.2 and 2.3 of the selection algorithm, given next.

ALGORITHM 4.6
(Selection)

Input: *An array* $A = (a_1, a_2, \ldots, a_n)$ *and a positive integer* k, *where* $1 \leq k \leq n$, *such that* $\log n$ *is an integer that divides* n.
Output: *An element* a_i *of* A *such that* $rank(a_i : A) = k$.
begin
1. *Set* $n_0 := n$; $s := 0$.
2. **while** $n_s > n/\log n$ **do**
   - *2.1. Set* $s = s + 1$.
   - *2.2. Partition* A *into blocks* B_i*'s, each consisting of* $\log n$ *consecutive elements of* A. *Compute the median* m_i *of each block* B_i.
   - *2.3. Compute the median* a *of* $(m_1, m_2, \ldots m_{\frac{n}{\log n}})$ *by using the pipelined merge-sort algorithm.*
   - *2.4. Determine the numbers* s_1, s_2, *and* s_3 *of the elements smaller than, equal to, and larger than* a, *respectively.*

2.5. Case statement:

$s_1 < k \leq s_1 + s_2$	*: Output a and* **exit.**
$k \leq s_1$	*: Compact those elements of A smaller than a into consecutive locations, and set* $n_s := s_1$.
$k > s_1 + s_2$	*: Compact those elements of A larger than a into consecutive locations, set* $n_s := s_3$ *and* $k = k - (s_1 + s_2)$.

3. Sort the array consisting of the remaining n_s elements. The kth element is the desired output.

end

EXAMPLE 4.12:

Let $A = (5, -30, -10, 7, 25, 16, 31, -20, 8, 9, -3, 5, 13, 17, 21, 19)$ and $k = 4$. The blocks B_i's are given by $B_1 = (5, -30, -10, 7), B_2 = (25, 16, 31, -20)$, $B_3 = (8, 9, -3, 5)$, and $B_4 = (13, 17, 21, 19)$. Hence, the medians are $m_1 = -10, m_2 = 16, m_3 = 5$, and $m_4 = 17$. The median of the m_i's is $a = 5$. It follows that $s_1 = 4, s_2 = 2$, and $s_3 = 10$. Therefore, the desired element is in $A_1 = (-30, -10, -20, -3)$, since $k \leq s_1$. The kth smallest element can now be obtained by sorting of the elements of A_1. This sort yields -3 as the desired element. □

Theorem 4.7: *The selection algorithm (Algorithm 4.6) correctly computes the kth smallest element of the input array A. This algorithm runs in* $O(\log n \log\log n)$ *time, using a linear number of operations.*

Proof: The correctness proof is straightforward and is left to the reader.

As for the running time, it can be estimated as follows. Consider an arbitrary iteration s of the **while** loop (step 2). We can implement step 2.2 by using the linear-time sequential algorithm for selection. Each block takes $O(\log n)$ sequential time for a total number of $O(n_s)$ operations. Using the pipelined merge-sort algorithm, step 2.3 takes $O(\log n)$ time, using a total of $O\left(\frac{n_s}{\log n} \times \log n\right) = O(n_s)$ operations.

Step 2.4 can be done as follows. Mark each element a_i with 1, 2, or 3, depending on whether $a_i < a$, $a_i = a$, or $a_i > a$. Using the prefix-sums algorithm (Algorithm 2.1), we can then obtain s_1, s_2, and s_3 easily. Hence, step 2.4 takes $O(\log n)$ time using a total of $O(n_s)$ operations. Similarly, step 2.5 takes $O(\log n)$ time using a linear number of operations.

Therefore, each iteration of the **while** loop takes $O(\log n)$ time, with $O(n_s)$ operations. The following claim implies that $O(\log \log n)$ iterations are sufficient to reduce the size of our list below $n/\log n$.

Claim: The sizes of the lists in two consecutive iterations satisfy $n_{s+1} \leq \frac{3n_s}{4}$.

Proof of Claim: Clearly, it is enough to show that $\frac{n_s}{4} \leq rank(a : A_s) \leq \frac{3n_s}{4}$, where A_s is the array at the beginning of iteration s, and $A_0 = A$. Since a is the median of the m_i's, a is larger than one-half of the m_i's. But m_i is larger than $(\log n)/2$ elements in B_i, for each i. Therefore, a is greater than or equal to $\frac{n_s}{2 \log n} \times \frac{\log n}{2} = \frac{n_s}{4}$ elements of A_s. We can show, in a similar fashion, the other side of the inequality.

Therefore, step 2 takes $O(\log n \log \log n)$ time, using a total of $O(\Sigma n_s) = O(n)$ operations. Step 3 takes $O(\log n)$ time, using $O(n)$ operations. Hence, the complexity bounds of the algorithm follow. □

PRAM Model: The two nontrivial operations used in Algorithm 4.6 are the pipelined merge sort and the prefix sums. Both of these operations can be done within the stated resources on the EREW PRAM model. Recall, however, that our description of the pipelined merge-sort algorithm (Algorithm 4.4) was for the CREW PRAM model only. □

4.6 *Lower Bounds for Comparison Problems

Lower bounds play an important role in the theory of algorithms. They give an indication on the smallest amount of resources that have to be expanded to solve a given problem. For example, a lower bound $T(n) \geq f(n))$ on the time it takes to solve a given problem P of size n implies that *any* algorithm to solve P will require $f(n)$ steps, regardless of how clever the algorithm is or what methods it uses. The fact that an algorithm has to examine each item of the input at least once to solve any but the most specialized problems implies that $T(n) = \Omega(n)$ for the problem's *sequential* computation. The corresponding **trivial lower bound** on the PRAM model is $\Omega(n/p)$, where p is the number of processors available. Establishing stronger lower bounds for parallel computations is usually much more difficult.

A known methodology for simplifying the development of nontrivial lower bounds is to impose a structure on the type of algorithms allowed. In most cases, the resulting classes of algorithms seem to include the only

reasonable candidate algorithms for the problems under consideration. For example, in sequential computations, the **binary decision-tree** model has been used to derive lower bounds for comparison problems, and the **straight-line** model (Section 1.3.1) has been used to derive lower bounds for algebraic computations. In each case, the class of algorithms considered seems to be natural and seems to capture the essential algorithmic features needed to solve the corresponding problems.

In this section, we introduce the *parallel comparison-tree model*, which seems to be well suited for studying comparison problems within the parallel framework. We then use this model to derive lower bounds on the number of parallel steps required to handle searching, computing the maximum, and merging.

4.6.1 PARALLEL COMPARISON TREES

Let A be an array of elements drawn from a linearly ordered set. The outcome of a comparison between two elements a_i and a_j of A could be one of the following: (1) $a_i < a_j$, (2) $a_i = a_j$, or (3) $a_i > a_j$. We are interested in comparison algorithms where, at each step of the algorithm, many comparisons can be made in parallel. The outcome of these comparisons will determine the set of comparisons to be made during the next step. We do not charge any cost to the process of obtaining any information that can be deduced from the outcome of the comparisons. A powerful model for such algorithms is the parallel comparison tree.

A **parallel comparison tree** of degree p is a 3^p-ary tree in which each internal node v represents a set of p comparisons performed between p pairs of elements (not necessarily distinct), and each of the branches out of v corresponds to one of the 3^p possible outcomes of the comparisons. The parameter p, also called the **degree of parallelism,** can depend in an arbitrary fashion on n. Each leaf of the tree represents a solution to the problem under consideration. The **number of parallel steps** is defined to be the height of the parallel comparison tree.

EXAMPLE 4.13:

Let $X = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\}$ be a set of elements drawn from a linearly ordered set. A parallel comparison tree of degree $p = 2$ to identify the maximum element of X is illustrated in Fig. 4.17. The notation $x_i : x_j$ indicates a comparison between x_i and x_j. The number of possible outcomes of the comparisons performed at the root node is nine. Only the leftmost path leading to either x_8 or x_6 being the maximum is shown. □

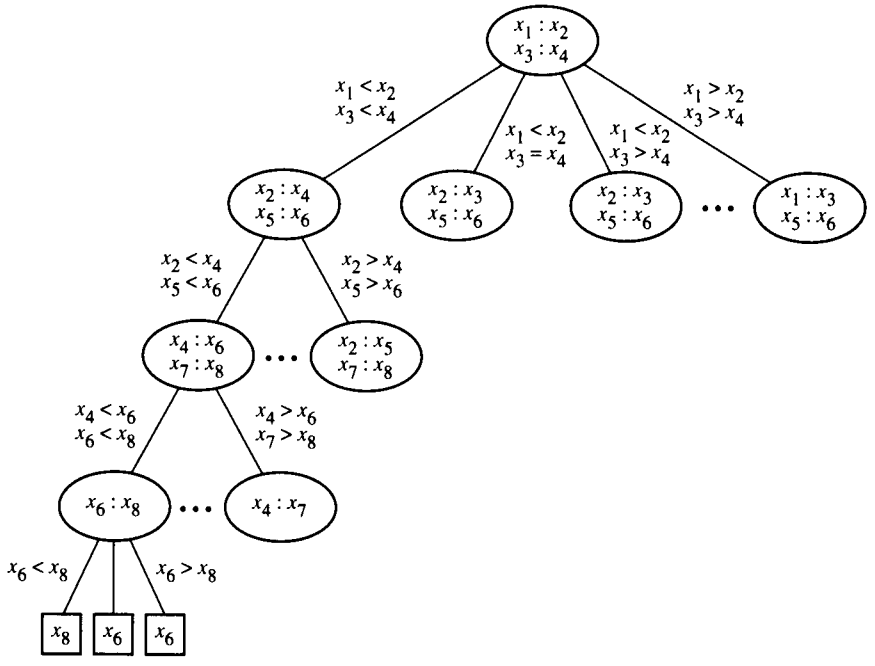


FIGURE 4.17
The leftmost path of a parallel comparison tree of degree 2 to compute the maximum of eight elements. Each internal node could have up to nine children corresponding to the nine possible outcomes of the two comparisons performed at the node.

The parallel comparison-tree model is a very powerful model for studying the parallel complexity of comparison problems. Note that, in this model, there are no restrictions on the selection of the comparisons made during any given step. Hence, no costs are charged to moving data around or to obtaining any information derived from the outcomes of the previous comparisons. For our purposes, a more reasonable model would be the comparison PRAM model introduced next.

The Comparison PRAM. The **comparison** PRAM model is a PRAM that allows only the operations of comparing elements and moving elements in the memory. No element can be viewed as a string of bits, and no arithmetic operations on the elements are allowed. Hence, a typical step of the comparison PRAM involves a set of comparisons made by some or all the available processors, followed by a set of data movements. Note that all the algorithms presented in this chapter satisfy the restrictions of the comparison PRAM model.

It is clear that each step of the comparison PRAM with p processors can be trivially simulated by a single step of a parallel comparison tree with degree p. Therefore, a lower bound for the parallel comparison-tree model will immediately hold for the comparison PRAM.

In the remainder of this section, we shall concentrate on establishing lower bounds on the parallel comparison-tree model.

Adversary Argument. Our lower-bound proofs use the following technique, called **adversary argument.** Given a parallel comparison tree T, the computation will follow a path from the root to a leaf such that the path depends on the particular input. The basic idea is to specify the input incrementally, as the algorithm progresses down the tree, such that the algorithm is forced to take a long path. In particular, for our class of problems, we do the following. After each parallel comparison step, an adversary will consistently specify the input such that the algorithm is forced to solve a large-sized instance—say, I—of the original problem; that is, we know nothing about the solution of I, and the desired solution of the original problem requires a solution of I. Solving the recurrence relation relating the sizes of the instances of two successive steps will produce the lower bound.



4.6.2 LOWER BOUNDS FOR SEARCHING

We start with the search problem considered in Section 4.1.

Theorem 4.8: *Given a sorted table X of n elements and an element y, the problem of searching for y in X requires $\Omega\left(\frac{\log n}{\log (p+1)}\right)$ parallel steps on the parallel comparison-tree model, where p is the degree of parallelism satisfying $p < n$.*

Proof: Since the table X is sorted, we can assume that each of the p comparisons carried out at a step involves y and an element of X. We establish the lower bound by using an adversary argument.

Let $x_{i_1}, x_{i_2}, \ldots, x_{i_p}$ be the elements of X used in the first parallel comparison step. Note that these elements do not need to be distinct. The x_{i_j}'s partition X into at most $p + 1$ intervals, each interval containing consecutive elements of X, excluding the x_{i_j}'s. One of these intervals—say, X_1—must contain $s_1 \geq (n - p)/(p + 1)$ elements of X. Since no comparisons have been made between any element of X_1 and y, the adversary chooses y to lie in X_1. Hence, we now have an arbitrary search problem on the table X_1 of size s_1.

This process can be continued such that the sizes s_i and s_{i+1} of the tables X_i and X_{i+1}, obtained after steps i and $i + 1$, respectively, are related by $s_{i+1} \geq (s_i - p)/(p + 1)$. Setting the initial condition $s_0 = n$, it is easy to verify

that the solution of this recurrence satisfies $s_i \geq \frac{n}{(p+1)^i} - 1$. Hence, the number t of parallel steps required satisfies $\frac{n}{(p+1)^t} - 1 \leq 1$, which implies that $t = \Omega\left(\frac{\log n}{\log (p+1)}\right)$. □

Corollary 4.6: *Searching a sorted table of n elements requires* $\Omega\left(\frac{\log n}{\log (p+1)}\right)$ *parallel steps on the comparison CRCW PRAM with p processors.*

4.6.3 LOWER BOUNDS FOR COMPUTING THE MAXIMUM

We now consider the problem of computing the maximum of n elements stored in an array $X = (x_1, x_2, \ldots, x_n)$.

We have already seen, in Section 2.6, that a doubly logarithmic-depth tree can be used to compute the maximum in $O(\log \log n)$ time on the common CRCW PRAM model, using a linear number of operations. We can rephrase this result as an $O(\log \log n)$ time algorithm using $n/\log \log n$ processors on the common CRCW PRAM model. We now show that the parallel time cannot be improved, even if we use n processors.

To establish our lower bound, we need the following theorem from *extremal graph theory*. For a given graph $G = (V, E)$, $U \subseteq V$ is an **independent set** if no two vertices of U are joined with an edge.

Theorem 4.9 (Turan): *Let* $G = (V, E)$ *be an arbitrary graph with n vertices and m edges. Then, G has an independent set of size at least* $n^2/(2m + n)$. □

Theorem 4.9 is actually a corollary to a classical result by Turan. Theorem 4.9 has a relatively simple proof, which is left to Exercise 4.29.

We are now ready to establish the lower bound on computing the maximum.

Theorem 4.10: *Computing the maximum of n elements in the comparison-tree model requires* $\Omega(\log \log n)$ *parallel steps, whenever the degree of parallelism is* $p \leq n$.

Proof: We prove by induction that, at the end of the $(i + 1)$st parallel step, an adversary can specify the input such that the maximum will lie in a set C_{i+1} of size c_{i+1}, with the following two properties:

1. No two elements of C_{i+1} have been compared before.
2. $c_{i+1} \geq \frac{c_i^2}{2p + c_i}$.

We start with the case $i = 0$, assuming $c_0 = n$. Hence, we consider the first parallel comparison step. This step involves p comparisons between pairs of elements from X. These comparisons can be represented by a graph $G = (V,E)$ as follows. The set V consists of n vertices such that v_i represents element x_i of X, where $1 \leq i \leq n$. The set E consists of all edges (v_i, v_j) such that one of the $p = n$ comparisons is between x_i and x_j. It follows that G has n vertices and at most p edges. By Theorem 4.9, G has an independent set—say, $C_1 = \{v_{i_1}, v_{i_2}, \ldots, v_{i_t}\}$—such that $t \geq n^2/(2p + n)$.

An adversary can choose the elements x_i's such that each x_{i_j} is larger than any of its adjacent vertices. This choice is clearly possible since C_1 is an independent set. It is obvious that C_1 satisfies the conditions of the induction hypothesis.

We can use exactly the same argument for an arbitrary step $i + 1$ by examining the possible effects of the p comparisons on C_i.

Using the preceding recurrence, it is clear that $c_{i+1} \geq c_i^2/3n$. Solving this recurrence, with the initial condition $c_0 = n$, we obtain that $c_i \geq n/(3^{2^i-1})$. The algorithm terminates when $c_i \leq 1$. Hence, the number of parallel steps is $\Omega(\log \log n)$. □

Corollary 4.7: *Computing the maximum of n elements requires* $\Omega(\log \log n)$ *time on the comparison PRAM with n processors.* □

The $\Omega(\log \log n)$ time lower bound obviously holds for the selection problem. In fact, this bound is achievable on the parallel comparison-tree model. Exercise 4.28 outlines an $O(\log \log^2 n)$ selection algorithm on this model. However, as we mentioned earlier (and as we shall show in Chapter 10), the selection problem requires $\Omega(\log n/\log \log n)$ time on the comparison CRCW PRAM, even with a polynomial number of processors. Therefore, *the parallel-comparison tree model is strictly more powerful than is the comparison PRAM model.*

4.6.4 LOWER BOUNDS FOR THE MERGING PROBLEM

The merging problem of two sorted sequences each of length n is considered next. Recall that the $O(\log \log n)$ time algorithm developed in Section 4.2 was based on a partitioning strategy that divided the original problem into $\sqrt{n}$ subproblems, each of which could then be solved independently. We shall use an adversary argument to show that, after each parallel step of a parallel comparison tree that solves the merging problem, the input can be specified such that $\Omega(\sqrt{n})$ independent subproblems, each of size $\Omega(\sqrt{n})$, have to be solved. This result will imply that $\Omega(\log \log n)$ time is needed to solve the merging problem.

To simplify the proof, we introduce the **(k, s) merging problem.** For $1 \leq i \leq k$, we are given k pairs of sequences (A_i, B_i) such that (1) each sequence A_i and B_i is of length s and is sorted in nondecreasing order, and (2) each element of $A_i \cup B_i$ is smaller than every element of $A_{i+1} \cup B_{i+1}$, for $1 \leq i \leq k-1$. The goal is to determine the sorted sequences C_i obtained by merging the pairs (A_i, B_i), for $1 \leq i \leq k$.

The crux of the lower-bound proof is given in the next lemma.

Lemma 4.10: *Suppose we are given a parallel comparison tree of degree p to solve the (k, s) merging problem, where $\frac{ks^2}{8} \geq p \geq 2ks$. After the first parallel step, an adversary can specify the input such that an arbitrary (k', s') merging problem will still have to be solved, where $k' \geq \frac{7}{4}\sqrt{\frac{pk}{2}}$ and $s' = \frac{s}{2}\sqrt{\frac{k}{2p}}$.*

Proof: Let $c = p/ks$. Partition each of A_i and B_i into $2\sqrt{2cs}$ blocks $\{A_{i,j}\}$ and $\{B_{i,j}\}$, where $1 \leq j \leq 2\sqrt{2cs}$. Note that $2\sqrt{2sc} \leq s$, since $c \leq s/8$ as $p \leq (ks^2)/8$.

For each $1 \leq i \leq k$, consider the Boolean array M_i of size $2\sqrt{2cs} \times 2\sqrt{2cs}$ defined as follows: $M_i(u, v) = 1$ if and only if at least one of the p comparisons made during the first parallel step is between an element in $A_{i,u}$ and an element in $B_{i,v}$.

Each M_i has $4\sqrt{2cs} - 1$ diagonals, including the main diagonal defined by $M_i(u,u)$, where $1 \leq u \leq 2\sqrt{2cs}$. Number these diagonals in an arbitrary way, and let d_{ij} be the number of 0 entries in the jth diagonal of M_i. Since only p comparisons were made during the first parallel step, the total number of 0 entries in all the M_i's is at least $8kcs - p = 7p$. This observation implies that

$$\sum_{i=1}^{k} \sum_{j=1}^{4\sqrt{2cs}-1} d_{ij} \geq 7p.$$

Interchanging the order of the summations, we obtain

$$\sum_{j=1}^{4\sqrt{2cs}-1} \left(\sum_{i=1}^{k} d_{ij} \right) \geq 7p.$$

Thus, there exists j' such that the corresponding inner term $\left(\sum_{i=1}^{k} d_{ij'}\right)$ is no smaller than the average. Therefore,

$$\sum_{i=1}^{k} d_{ij'} \geq \frac{7p}{4\sqrt{2cs}} = \frac{7}{4}\sqrt{\frac{pk}{2}}.$$

Now consider the j'th diagonal of M_i. Let the 0 entries correspond to the indices $(l_1, m_1), (l_2, m_2), \ldots, (l_{d_{ij'}}, m_{d_{ij'}})$ such that $l_1 < l_2 < \ldots < l_{d_{ij'}}$ and $m_1 < m_2 < \ldots < m_{d_{ij'}}$. Clearly, this ordering is possible, since the entries lie off the j'th diagonal of M_i. Consider the pairs (A_{i,l_j}, B_{i,m_j}), for $1 \leq j \leq d_{ij'}$. An adversary can specify each pair (A_i, B_i) such that each element of $A_{i,l_j} \cup B_{i,m_j}$ is smaller than every element of $A_{i,l_{j+1}} \cup B_{i,m_{j+1}}$.

Therefore, these new pairs form a new merging problem consisting of at least $k' = \frac{7}{4}\sqrt{\frac{pk}{2}}$ subproblems, each of size $s' = \frac{s}{2\sqrt{2cs}} = \frac{s}{2}\sqrt{\frac{k}{2p}}$; hence, the lemma follows. □

We are now ready to prove the lower bound on the (k, s) merging problem.

Theorem 4.11: *Let $T(k, s, p)$ be the number of parallel steps used by any parallel comparison tree with degree p to solve the (k, s) merging problem, where $2ks \leq p \leq \frac{ks^2}{8}$. Then, $T(k, s, p) = \Omega\left(\log\log s - \log\log\left(\frac{p}{ks}\right)\right)$.*

Proof: Using Lemma 4.10, it is clear that the following recurrence holds:

$$T(k, s, p) \geq 1 + T\left(\frac{7}{4}\sqrt{\frac{pk}{2}}, \frac{s}{2}\sqrt{\frac{k}{2p}}, p\right).$$

We claim the following, which will establish the lower bound:

$$T(k, s, p) \geq \frac{1}{4}\log\left(\frac{\log\left(\frac{p}{k}\right)}{\log\left(\frac{p}{ks}\right)}\right). \tag{4.1}$$

To verify this claim, we will use induction on k and s. Hence, we can assume that

$$T\left(\frac{7}{4}\sqrt{\frac{pk}{2}}, \frac{s}{2}\sqrt{\frac{k}{2p}}, p\right) \geq \frac{1}{4}\log\frac{\log\frac{4}{7}\sqrt{\frac{2p}{k}}}{\log\frac{16}{7}\frac{p}{ks}}.$$

Note that $\log\frac{4}{7}\sqrt{\frac{2p}{k}} \geq \frac{1}{4}\log\frac{p}{k}$ (assuming, without loss of generality, that $s \geq 2$ and hence $\frac{p}{k} \geq 4$). We also have that $\log\frac{16}{7}\frac{p}{ks} \leq 3\log\frac{p}{ks}$. It follows that

$$\begin{aligned}
T(k, s, p) &\geq 1 + \frac{1}{4}\log\frac{\log\left(\frac{4}{7}\sqrt{\frac{2p}{k}}\right)}{\log\left(\frac{16}{7}\frac{p}{ks}\right)} \\
&\geq 1 + \frac{1}{4}\log\frac{1}{12}\frac{\log\frac{p}{k}}{\log\frac{p}{ks}} \\
&\geq 1 + \frac{1}{4}\log\left(\frac{1}{12}\right) + \frac{1}{4}\log\frac{\log\frac{p}{k}}{\log\frac{p}{ks}} \\
&\geq \frac{1}{4}\log\frac{\log\frac{p}{k}}{\log\frac{p}{ks}}.
\end{aligned}$$

The last inequality holds because $1 + (1/4)\log(1/12) > 0$. Therefore, Equation 4.1 holds.

Since, by assumption, $2s \leq \frac{p}{k} \leq \frac{s^2}{8}$, the preceding inequality implies that $T(k, s, p) = \Omega\left(\log \log s - \log \log \frac{p}{ks}\right)$, and the proof of the theorem is complete. □

Setting $k = 1$ and $s = n$, we obtain the following corollary.

Corollary 4.8: *The merging problem of two sorted sequences, each of length n, requires* $\Omega(\log \log n)$ *time on the comparison CRCW PRAM whenever the number p of processors satisfies* $p = O(n \log^r n)$, *for any positive constant r.* □

4.6.5 LOWER BOUNDS FOR SORTING

As for the sorting problem, it is well known that $\Omega(n \log n)$ comparisons are required to sort n numbers on the binary decision-tree model. Hence, $\Omega(\log n)$ time is required on any comparison PRAM, whenever the number p of processors is $\leq n$. The case when $p > n$ is more interesting, and is addressed in Exercises 4.23, 4.26, and 4.27. The bibliographic notes provide related references for the interested reader.

4.7 Summary

In this chapter, we developed efficient parallel strategies to handle several fundamental comparison problems. These strategies include parallel searching, partitioning, and pipelined divide-and-conquer. We showed that the performance of the resulting parallel algorithms for searching, computing the maximum, and merging cannot be improved even on the strong parallel comparison-tree model. This chapter also provided a brief introduction to the class of sorting networks by presenting the bitonic sorting algorithm and the construction of bitonic sorting networks. The important odd-even sorting network and permutation networks are sketched in Exercises 4.33, 4.37, and 4.38.

A summary of the main results is given in Tables 4.1 and 4.2. Table 4.1 lists the performance of the algorithms described, and Table 4.2 lists the lower bounds established on the parallel comparison-tree model (hence, lower bounds that hold on the comparison CRCW PRAM model). The parameter p appearing in the tables is the number of processors.

TABLE 4.1
ALGORITHMS INTRODUCED IN THIS CHAPTER.

Algorithm		Section	Time*	PRAM Model
4.1	Parallel Search	4.1	$O\left(\frac{\log (n+1)}{\log (p+1)}\right)$	CREW
4.2	Ranking a Sorted Sequence in Another Sorted Sequence	4.2.2	$O\left(\frac{(n+m)\log\log m}{p} + \log\log m\right)$	CREW
	Optimal Merging	4.2.3	$O\left(\frac{n}{p} + \log\log n\right)$	CREW
4.3	Simple Merge Sort	4.3.1	$O\left(\frac{n\log n}{p} + \log n \log\log n\right)$	CREW
4.4	Pipelined Merge Sort	4.3.2	$O\left(\frac{n\log n}{p} + \log n\right)$	CREW
4.5	Bitonic Merge	4.4	$O\left(\frac{n\log n}{p} + \log n\right)$	EREW
4.6	Selection	4.5	$O\left(\frac{n}{p} + \log n \log\log n\right)$	EREW

*p: number of processors.

TABLE 4.2
LOWER BOUNDS ESTABLISHED ON THE PARALLEL-COMPARISON TREE MODEL.

Problem	Lower Bound	Number of Processors
Searching	$\Omega\left(\frac{\log n}{\log (p+1)}\right)$	$p \leq n$
Maximum	$\Omega\left(\frac{n}{p} + \log\log n\right)$	$p \leq n$
Merging	$\Omega\left(\frac{n}{p} + \log\log n\right)$	$p \leq n \log^k n$
		Any constant k

Exercises

4.1. Let X be an arbitrary sequence of elements from a linearly ordered set S, and let $y \in S$. Suppose that we want to determine all the occurrences, if any, of y in X. Develop an $O(\log n)$ time EREW PRAM algorithm to solve this problem, using a linear number of operations.

4.2. Prove that the parallel-search algorithm (Algorithm 4.1) is correct.

4.3. Will Algorithm 4.1 work correctly on the CREW PRAM model if the sorted elements of X are not necessarily distinct? Which occurrence of y, if any, will be identified?

4.4. Let X be a sorted sequence of n distinct elements, and let y be an arbitrary element. Show that, given any nondecreasing function $t(n)$, where $1 \leq t(n) \leq \log n$, we can perform the search for y in X in $O(t(n))$ time, using a total of $O(t(n) \cdot n^{\frac{1}{t(n)}})$ operations.

4.5. Show how to solve the search problem corresponding to a sorted table of n entries in $O(\log n - \log p)$ steps on the EREW PRAM model with p processors, provided that the search key can be accessed concurrently by all the processors.

4.6. Given a monotonic binary sequence $X = (x_1, x_2, \ldots, x_n)$ (that is, X consists of a subsequence of zeros followed by a subsequence of ones), develop an EREW PRAM algorithm to determine the number of zeros assuming there are p processors available, where $1 \leq p \leq \sqrt{n}$.

4.7. Let $B = (b_1, b_2, \ldots, b_n)$ be a sorted table of distinct elements, and let $A = (a_1, a_2, \ldots, a_k)$ be an arbitrary sequence of elements such that $k = O(\log n)$. Develop an algorithm to solve the search problem of each a_i in the table B in $O(\log n)$ time using k processors on the EREW PRAM.

4.8. Let X be a sorted set of n keys, and let y be an arbitrary element. Develop a procedure to search for y in X that runs in $O(\log n/\log\log n)$ time on the comparison EREW PRAM model, assuming that local computations can be performed at no cost. Note that accessing the search key y by p processors on the EREW PRAM requires $\Omega(\log p)$ steps. *Hint:* Generalize the binary search tree to a multiway-search tree such that each node at level i contains $i + 1$ keys. Each such node is associated with a single processor.

4.9. We are interested in solving the search problem corresponding to a sorted table of size n on the EREW PRAM with the following additional assumptions. An arbitrary number of keys can be encoded and then stored in a single location. Once such an encoding is stored in a location, a single processor can access it in 1 unit of time. Assume, as in Exercise 4.8, that local operations can be performed for free.

Show that the search problem can be solved in $O(\sqrt{\log n})$ steps in this model. *Hint:* Use a multiway-search tree. Store the encoding of the tuple of keys at each node in one memory location.

4.10. The procedure referred to in Lemma 4.1 to rank a short sequence in a sorted sequence requires that the elements of X be distinct. Assume that the elements are not necessarily distinct. Can you still obtain the same performance? Explain your answer.

4.11. Algorithm 4.2 was presented in the WT presentation framework. Show how to handle the corresponding processor allocation problem to implement this algorithm on a CREW PRAM with $n + m$ processors.

Hint: Store the $j(i)$'s in a separate array. The pair (B_i, A_i) should be handled by processors $P_{i\sqrt{m}+j(i)+1}, \cdots, P_{(i+1)\sqrt{m}+j(i+1)}$. Decompose the $n + m$ processors into groups, each with $\sqrt{m}$ processors. Let P_{i_l} be the first processor in the lth group. The kth processor in group l can determine whether or not P_{i_l} is assigned to the subproblem (B_k, A_k).

4.12. Consider the problem of merging two sequences A and B of lengths n and m, respectively, where $m \leq n$. Assume that all the entries are distinct. Develop a fast optimal parallel algorithm similar to the one developed in Section 4.2.3.

4.13. Provide a solution to the processor allocation problem corresponding to the algorithm for merging in the presence of a cover, outlined in Lemma 4.3, on the PRAM model with $n + m$ processors. Which PRAM model do you need?

4.14. Consider the problem of merging two sorted lists of lengths m and n, where $m \leq n$. Develop a merging algorithm that runs in time $O\left(\left\lceil \frac{\log m}{\log \frac{p}{n}} \right\rceil\right)$, where p is the number of processors used, and $p \geq n$. Specify the comparison PRAM model you use.

4.15. The ANSV problem was introduced in Exercise 2.14. For the sake of completeness, we introduce it again.

Let $A = (a_1, a_2, \ldots, a_n)$ be an array whose elements are drawn from a linearly ordered set. The *left match* of a_i, if it exists, is the nearest element $a_{l(i)}$ to the left of a_i satisfying $a_{l(i)} < a_i$. We can define the *right match* $a_{r(i)}$ of a_i, if it exists, in a similar fashion. The ANSV problem corresponding to A is to determine $a_{l(i)}$ and $a_{r(i)}$ for each $1 \leq i \leq n$, whenever they exist.

Show how to reduce the merging of two sorted sequences of lengths n and m to the ANSV problem corresponding to an array of length $n + m$.

4.16. Using a balanced binary tree, develop an $O(\log n)$ time algorithm to solve the ANSV problem (see Exercise 4.15) of an array of length n. What is the total number of operations used? Which PRAM model do you need?

4.17. This exercise sketches a method to solve the ANSV problem (see Exercise 4.15), using a linear number of operations. Let A be an array of length n.

a. Let $I[j] = \{j + 1, \ldots, r(j) - 1\}$, whenever $r(j)$ exists, where $1 \leq j \leq n$. Show the following:

- If $k \in I[j]$, then $r(k), l(k) \in I[j] \cup \{j, r(j)\}$.
- If $k \notin I[j]$, then $r(k), l(k) \notin I[j]$.

b. Let $I'[j] = \{l(j) + 1, \ldots, j - 1\}$, whenever $l(j)$ exists. Prove that $I'[j]$ satisfies properties similar to those satisfied by $I[j]$.

c. Partition A into $n/\log n$ blocks, say $A_1, A_2, \ldots$, and let $m(i)$ be the index of the minimum element in A_i. Prove that, if the right match of $a_{m(i)}$ belongs to $A_{i'}$, $i' \neq i + 1$, then there exists a unique k, $i < k < i'$, such that the left match of $a_{m(k)}$ is in A_i and the right match of $a_{m(k)}$ is in $A_{i'}$.

d. *Based on the properties shown in parts a, b, and c, develop an $O(\log n)$ time algorithm to solve the ANSV problem such that the total number of operations used is $O(n)$. Your algorithm should run on the CREW PRAM model. *Hint:* Reduce ANSV to a set of disjoint merging problems.

e. *Develop an optimal $O(\log \log n)$ time algorithm to solve the ANSV problem. Your algorithm should run on the common CRCW PRAM model.

4.18. Suppose that, for a binary tree $T = (V, E)$, where $V = \{1, 2, \ldots, n\}$, we are given two arrays:

1. $P = (p_1, p_2, \ldots, p_n)$ containing the numbering of the nodes in a preorder traversal
2. $I = (i_1, i_2, \ldots, i_n)$ containing the numbering of the nodes in inorder traversal

Develop an $O(\log \log n)$ time algorithm that generates, for each node i, i's left child and i's right child, whenever they exist. Your algorithm should use $O(n)$ operations. *Hint:* You can assume the existence of an $O(\log \log n)$ time ANSV algorithm that uses a linear number of operations (see Exercise 4.17).

4.19. The parentheses-matching problem was introduced in Exercise 3.31. Using the optimal $O(\log \log n)$ time ANSV algorithm (see Exercise 4.17), show how to solve the parentheses-matching problem in $O(\log \log n)$ time, using a total of $O(n)$ operations.

4.20. An important sorting strategy, **quicksort,** consists of (1) picking an element x from the sequence X to be sorted, (2) partitioning X into the two subsequences $X_1 = \{y \in X \mid y \leq x\}$ and $X_2 = \{y \in X \mid y \geq x\}$, and (3) recursively sorting X_1 and X_2.

a. Develop a nonrecursive version of quicksort, assuming that the element x is chosen to be the last element in the array under consideration. Assuming that there are p processors available, where $p \leq n$, what is the running time of your algorithm?

b. Suppose that we use the parallel-selection algorithm presented in the text to identify x as the median of X. What is the running time of the corresponding sorting algorithm on the PRAM model? Is the algorithm optimal? Specify your PRAM model.

4.21. Rewrite Algorithm 4.3 such that only $O(n)$ space is used.

4.22. Establish the correctness of Lemma 4.5.

4.23. *Suppose we are interested in sorting a sequence of n elements on the CRCW PRAM with p processors, where $2n \leq p \leq n^2$. Show how to modify the pipelined merge-sort algorithm to obtain such an algorithm that runs in $O\left(\frac{\log n}{\log \log \frac{2p}{n}}\right)$ time. *Hint:* Let $r = \frac{p}{n}$. Make the binary merge-sort tree into an r-ary merge tree, and redefine $Sample(L_s[v])$ to be every r^2th item, or rth item, or every item.

4.24. Establish the correctness of Algorithm 4.5.

4.25. Develop a parallel comparison-tree algorithm to merge two sorted sequences of lengths n and m in $O((\log \log n - \log \log r))$ parallel steps, where the degree of parallelism is $\lfloor r\sqrt{nm} \rfloor$, $r \geq 2$ and $n \leq m$. *Hint:* Consider the elements $i\lceil\sqrt{\frac{n}{r}}\rceil$ in X and $i\lceil\sqrt{\frac{m}{r}}\rceil$ in Y, $i = 1, 2, \ldots$.

4.26. Using the fact that there exists a sorting network of n elements with depth $O(\log n)$ and size $O(n \log n)$, show that sorting n elements can be done in $O\left(\frac{\log n}{\log\left(1+\frac{p}{n}\right)}\right)$ parallel steps on the parallel comparison-tree model of degree $p \geq n$. *Hint:* Shrink each $t = \frac{1}{2}\log\left(1 + \frac{p}{n}\right)$ steps into a single step on the parallel comparison-tree model.

4.27. Let $c(k, n)$ be the minimum total number of comparisons required to sort any n elements in k parallel steps on the parallel comparison-tree model.

a. Show that $c(k, n) \geq c(k, n - x) + n - x + c(k - 1, x)$, for some $0 < x \leq n$. *Hint:* Consider the graph $G = (V, E)$, where $|V| = n$, and E represents the comparisons made during the first parallel step. Let S be a maximal independent set.

b. *Using the recursion in part a, show that $c(k, n) > k\left(\frac{n^{1+\frac{1}{k}}}{e} - n\right)$.

c. Compare the result of part b with the upper bound established in Exercise 4.26.

4.28. You wish to find the kth smallest element of an array $A = (a_1, a_2, \ldots, a_m)$ on the parallel comparison-tree model with degree n. An element a_l of A will be called a *lower approximation* to the kth smallest element if

$$k - \frac{m\sqrt{m}}{4\sqrt{n}} \leq rank(a_l : A) \leq k.$$

Similarly, a_u is an *upper approximation* if

$$k + \frac{m\sqrt{m}}{4\sqrt{n}} \geq rank(a_u : A) \geq k.$$

A possible strategy to handle the selection problem is to identify a lower and an upper approximation that can be used to restrict the subset of possible candidates. Repeat this process until the desired element is

identified. We define a *stage* to be the process of identifying the approximations and shrinking the set accordingly.

a. Show that $O(\log \log n)$ stages are sufficient to identify the kth smallest element.

b. *Develop an $O(\log \log n)$ step algorithm to identify a lower and an upper approximation. *Hint:* Divide the current set S into blocks, each of r^2 elements, where $r = \lceil \sqrt{\frac{cn}{m}} \rceil$, for some constant $c > 0$. Let s be the number of blocks. Sort each block and form the set T consisting of the (ri)th items, for $i = 1, 2, \ldots, r$.

4.29. *Prove Turan's theorem (Theorem 4.9). *Hint:* Construct an independent set I as follows. Let v_1 be a vertex of minimum degree. Put v_1 in I and remove v_1 and all the vertices adjacent to it. Repeat the process on the remaining graph.

4.30. Consider the bitonic sorting network developed in Section 4.4 to sort bitonic sequences. As in the text, assume that the length n of the input is a power of 2—say, $n = 2^k$. This network can be viewed as a set of n parallel horizontal lines, numbered $0, 1, \cdots, n-1$, such that each stage consists of a set of $\frac{n}{2}$ comparators connecting pairs of the horizontal lines.

Show that the comparisons made during stage i are between lines whose indices differ in the ith most significant bit, where $1 \leq i \leq k$.

4.31. Show that an n-input comparator network is a sorting network if and only if it sorts correctly all sequences of zeros and ones. This fact is usually referred to as the **zero-one principle.**

4.32. Use the zero-one principle, stated in Exercise 4.31, to show that the bitonic sorting network sorts bitonic sequences correctly.

4.33. Let $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_n)$ be two sorted sequences such that n is a power of 2. The **odd-even merge** algorithm consists of recursively merging the two "odd" sequences, $(a_1, a_3, \ldots, a_{n-1})$ and $(b_1, b_2, \ldots, b_{n-1})$, into $(c_1, c_2, \ldots, c_n)$, and merging the two "even" sequences, $(a_2, a_4, \ldots, a_n)$ and $(b_2, b_4, \ldots, b_n)$, into $(c'_1, c'_2, \ldots, c'_n)$. Then, the sorted sequence is given by $(c_1, \min\{c'_1, c_2\}, \max\{c'_1, c_2\}, \min\{c'_2, c_3\}, \max\{c'_2, c_3\}, \ldots, \min\{c'_{n-1}, c_n\}, \max\{c'_{n-1}, c_n\}, c'_n)$.

a. Using the zero-one principle, show that the odd-even merge algorithm works correctly.

b. Derive a sorting network based on the odd-even merge algorithm. State its depth and its size.

4.34. Consider a bitonic sequence X of length $n = 2^k$, k a positive integer. Assume without loss of generality that all the elements of X are distinct.

a. Show that the horizontal cut with the unique cross-over property can be determined in at most k comparisons.

b. Deduce that $L(X)$ and $R(X)$ can be determined in at most k comparisons.

c. Show that the results of the comparisons made by the bitonic sorting network are uniquely determined by the results of $2n - k - 2$ comparisons.

4.35.* a. Use Exercise 4.34 to develop an $O(\log n)$ time EREW PRAM algorithm to sort a bitonic sequence. The total number of operations must be linear. *Hint:* Each stage of the bitonic sorting algorithm may now take $\log n$ steps (instead of a single parallel step). Examine the possibility of pipelining these stages.

b. Deduce an EREW PRAM sorting algorithm that runs in $O(\log^2 n)$ time, using $O(n \log n)$ operations.

4.36. This exercise suggests a simple optimal $O(\log n)$ time EREW PRAM algorithm for merging two sorted sequences A and B, each of length n.

We start by noting that the bitonic sorting network provides an EREW PRAM merging algorithm that runs in $O(\log n)$ time, but also uses $O(n \log n)$ operations. To make this algorithm optimal, we start by partitioning A and B into $\frac{n}{\log n}$ blocks of almost equal lengths. We define the subarrays A' and B' as in the optimal merging algorithm presented in the text. We merge A' and B' by using the bitonic sorting algorithm. Now we have to rank each element of A' in B (and each element of B' in A). For this ranking, several elements of A' may need to access the same block of B.

Resolve the memory conflicts by using the pipelining technique. Deduce an optimal $O(\log n)$ time merging algorithm that runs on the EREW PRAM.

4.37. A **switching network** is a network made up of **switches,** where a switch swaps its two inputs if it is on, and passes its two inputs through if it is off. The size and the depth of a switching network can be defined as in the case of comparator networks. A switching network, with n inputs $x_1, x_2, \cdots, x_n$ and n outputs $y_1, y_2, \cdots, y_n$, is called a **permutation network,** if for every permutation $\pi \in S_n$, the switches can be set such that $x_{\pi(1)}, x_{\pi(2)}, \cdots, x_{\pi(n)}$ appear on the output lines in this order.

a. Assuming that $n = 2^k$, show that the network defined recursively in Fig. 4.18 is a permutation network. This network is called **Waksman's permutation network.**

b. Determine the size and the depth of this network.

4.38. The following general construction of permutation networks was developed by Clos [10], and the resulting networks are called **Clos networks.** Let $n = mk$. Suppose that we know how to construct permutation networks on m and k inputs, denoted by $CL(m)$ and $CL(k)$, respectively.

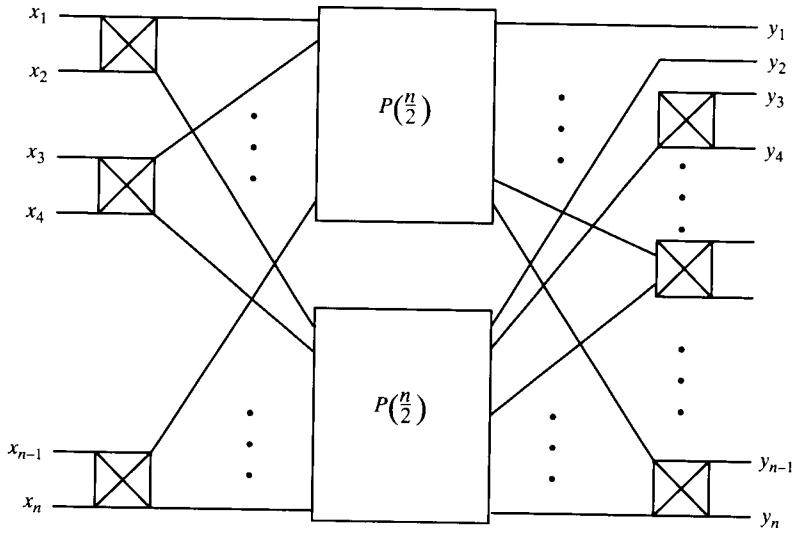


FIGURE 4.18
Waksman's permutation network. The switches can be set such that any permutation of the x_i's can appear on the output lines.

a. Show that the three-stage network shown in Fig. 4.19 is a permutation network on n inputs. *Hint:* Use induction on m and Hall's theorem on distinct representatives.

b. Consider the case where $m = 2$. Apply the construction recursively to the middle network, assuming that n is a power of 2. The corresponding network is called a **Benes network** [6]. What are the size and the depth of the corresponding permutation networks?

Bibliographic Notes

Snir [18] presents several results on parallel searching on the EREW PRAM model. Exercises 4.6, 4.8, and 4.9 are taken from there. The partitioning strategy to handle the merging problem in $O(\log \log n)$ time was developed by Valiant [19] on the parallel comparison-tree model introduced in the same paper. An optimal $O(\log \log n)$ time merging algorithm and related results were later described in [16]. The pipelined merge-sort algorithm was developed by Cole [12]; this algorithm is commonly referred to as *Cole's merge-sort algorithm* in the literature. The solution to Exercise 4.23 can also be found in [12]. The generalized version of the pipelined merge-sort algorithm described in the text appeared in [3]. Sorting n elements on the parallel comparison-tree model with degree $p \geq n$ was studied in [4] where the solutions to Exercises 4.26 and 4.27 can be found. Bitonic sorting networks and those based on the odd-even merge algorithm (Exercise 4.33) were described by Batcher

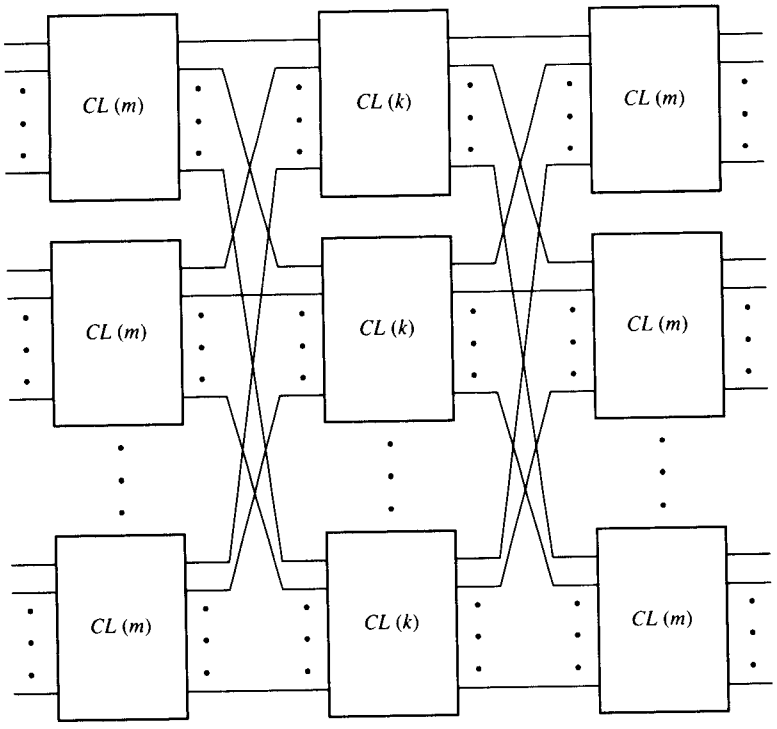


FIGURE 4.19
A general construction of a Clos network corresponding to $n = mk$ using the building blocks $CL(m)$ and $CL(k)$, which are permutation networks on m and k inputs, respectively.

[5]. The AKS sorting network, which is the first sorting network of depth $O(\log n)$ and size $O(n \log n)$, was introduced in [2]. The selection algorithm presented in Section 4.5 was developed by Vishkin [20]. An optimal $O(\log n \log^* n)$ time algorithm running on the EREW PRAM appeared in [11]. This paper contains a modification of the algorithm that runs optimally in $O(\log n \log^* n / \log \log n)$ time on the CRCW PRAM. On the parallel comparison-tree model, the $O((\log \log n)^2)$ time algorithm (Exercise 4.28) appeared in [13], and an $O(\log \log n)$ time algorithm was presented in [1]. The $\Omega(\log \log n)$ lower bound for computing the maximum of n elements was described by Valiant [19], and the lower bound for merging was presented by Borodin and Hopcroft [9]. This latter paper contains a solution to the processor allocation problem corresponding to the $O(\log \log n)$ time merging algorithm (Exercise 4.11).

Exercise 4.14 is taken from [17]; Exercises 4.15 through 4.19 are from [7]. The proof of Turan's Theorem as suggested in Exercise 4.29 was communicated to the author by U. Vishkin. The zero-one principle appears in Knuth [15]. The two $O(\log n)$ time EREW PRAM algorithms for merging (Exercises 4.35 and 4.36) appeared in [8] and [14], respectively. A description of Waksman's permutation network was given in [21].

References

1. Ajtai, M., J. Komlos, W. L. Steiger, and E. Szemeredi. Optimal parallel selection has complexity $O(\log \log n)$. *Journal of Computer and System Sciences* 38(1):125-133, 1989.
2. Ajtai, M., J. Komlos, and E. Szemeredi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3(1):1-19, 1983.
3. Atallah, M., R. Cole, and M. Goodrich. Cascading divide-and-conquer: A technique for designing parallel algorithms. *SIAM J. Computing*, 18(3):499-532, 1989.
4. Azar, Y., and U. Vishkin. Tight comparison bounds on the complexity of parallel sorting. *SIAM J. Computing*, 16(3):458-464, 1987.
5. Batcher, K. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, Atlantic City, NJ, 1968, pages 307-314.
6. Benes, V. E. *Mathematical Theory of Connecting Networks and Telephone Traffic.* Academic Press, New York, 1965.
7. Berkman, O., B. Schieber, and U. Vishkin. Some doubly logarithmic optimal parallel algorithms based on finding all nearest smaller values. Technical Report UMIACS-TR-88-79, Institute for Advanced Computer Studies, University of Maryland, College Park, MD, 1988.
8. Bilardi, G., and A. Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines. *SIAM J. Computing*, 18(2):216-228, 1989.
9. Borodin, A., and J. Hopcroft. Routing, merging, and sorting on parallel models of computation. *Journal of Computer and System Sciences*, 30(1):130-145, 1985.
10. Clos, C. A study of non-blocking switching networks. *Bell Systems Technical Journal*, 32(2):406-425, 1953.
11. Cole, R. An optimally efficient selection algorithm. *Information Processing Letters*, 26(6):295-299, 1988.
12. Cole, R. Parallel merge sort. *SIAM J. Computing*, 17(4):770-785, 1988.
13. Cole, R., and C. Yap. A parallel median algorithm. *Information Processing Letters*, 20(3):137-139, 1985.
14. Hagerup, T., and C. Rub. Optimal merging and sorting on the EREW PRAM. *Information Processing Letters*, 33(4):181-185, 1989.
15. Knuth, D. *Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.
16. Kruskal, C. Searching, merging, and sorting in parallel computation. *IEEE Transactions on Computers*, C-32(10):942-946, 1983.
17. Shiloach, Y., and U. Vishkin. Finding the maximum, merging and sorting in a parallel computation model. *Journal of Algorithms*, 2(1):88-102, 1981.
18. Snir, M. On parallel searching. *SIAM J. Computing*, 14(3)688-707, 1985.
19. Valiant, L. G. Parallelism in comparison problems. *SIAM J. Computing*, 4(3):348-355, 1975.
20. Vishkin, U. An optimal parallel algorithm for selection. *Advances in Computing Research*, JAI Press Inc., Greenwich, CT, 1987.
21. Waksman, A. A permutation network. *JACM*, 15(1):159-163, 1968.

5

Graphs

Graphs can be used to represent relationships occurring in many real-world situations, such as communication systems, electrical networks, transportation systems, and scheduling systems; moreover, they can be used as a tool to structure such relationships, as in the development of efficient data structures. Processing graphs efficiently has been a research focus of algorithms designers. Algorithmic graph theory is a rich area that contains many beautiful results. In this chapter, we introduce efficient parallel algorithms to handle a few representative graph problems. Research in parallel graph algorithms is currently being pursued vigorously, and significant improvements over the algorithms presented here are likely to be made in the near future. However, our emphasis will be on developing a few basic techniques that have general applicability in handling graph problems.

We begin in Section 5.1 by presenting two efficient parallel algorithms to identify the connected components of a graph. The resource requirements for solving this problem will determine the complexities of most of the algorithms covered in this chapter. One of our connected-components algorithms is then extended to handle the problem of the minimum spanning tree (Section 5.2). Another connectivity problem, that of identifying the blocks of a graph, is reduced to the connected-components problem in Section 5.3. In addition to these connectivity problems, we present a search method based

on the ear decomposition of a graph (Section 5.4), and parallel algorithms for two important problems in directed graphs—namely, transitive closure and all pairs shortest paths (Section 5.5).

5.1 Connected Components

Let $G = (V, E)$ be an undirected graph with $|V| = n$ and $|E| = m$. Two vertices u and v are *connected* if $u = v$ or there exists a path $P = (u = x_1, x_2, \ldots, x_k = v)$, such that $(x_i, x_{i+1}) \in E$, where $1 \leq i \leq k - 1$. This relation is an equivalence relation on V, and hence partitions V into equivalence classes $\{V_j\}_{j=1}^{l}$. The subgraphs $G_j = (V_j, E_j)$, where $E_j = \{(x, y) \in E \mid x, y \in V_j\}$, are called the **connected components** of G.

EXAMPLE 5.1:

Consider the graph shown in Fig. 5.1. This graph has three connected components, each of which is shown separately. □

One of the most basic problems in graph theory is to determine the connected components of a graph. A simple sequential algorithm—say, one based on depth-first search—runs optimally in $O(n + m)$ time. The **depth-first search** of a graph $G = (V, E)$ is a systematic method of visiting all the

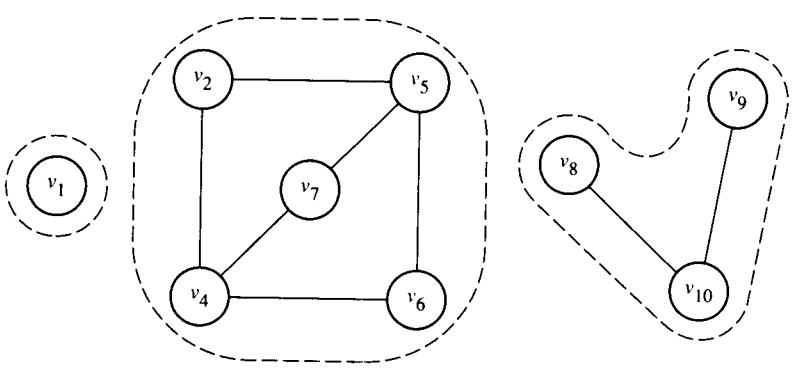


FIGURE 5.1
A graph whose three connected components are each enclosed within a dashed closed curve.

vertices of G starting with a vertex $v \in V$ and using edges from G. A sketch of this search method is given next.

Let k denote the label of a connected component. Initially, we set $k = 1$. The search procedure applied to vertex v begins by visiting v and labeling it with the value of k. We then pick an unlabeled vertex w adjacent to v, and apply the search procedure to w. Once the search procedure at w terminates, we backtrack to v and apply the search procedure to an unlabeled vertex adjacent to v. If no such vertex exists, the search procedure at v terminates. We then increment the connected-component label by setting $k = k + 1$, and pick an arbitrary unlabeled vertex $v' \in V$. The search procedure is now applied to v'. The algorithm terminates when all the vertices have been labeled. The vertices belonging to the same connected component receive the same label.

Developing an efficient parallel algorithm turns out to be a more challenging problem. In this section, we present two parallel algorithms that solve the connected-components problem. The first algorithm is more appropriate whenever the graph is given by its adjacency matrix; the second is superior if the input is given as a set of edges and the number m of edges satisfies $m = o(n^2/\log n)$, where n is the number of vertices in the graph.

5.1.1 OVERALL STRATEGY

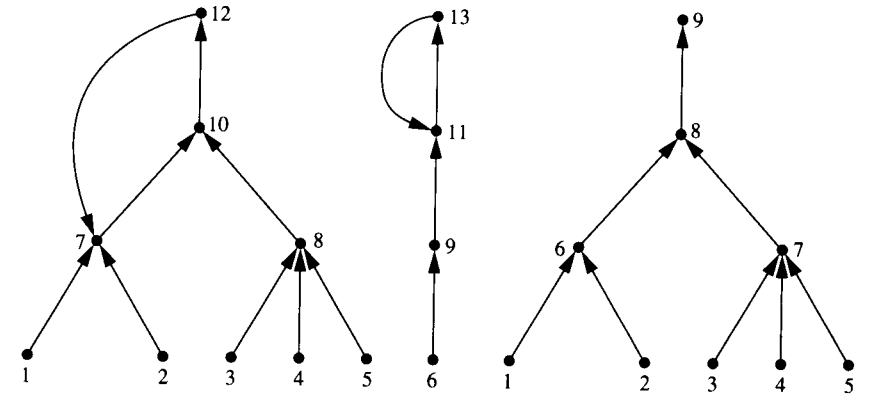
We begin with a few definitions. A **pseudoforest** is a directed graph in which each vertex has an outdegree less than or equal to 1 (also introduced in Exercise 2.42). Notice that an *arbitrary function* $D : V \rightarrow V$ defines a pseudoforest (V, F), where $F = \{\langle v, D(v)\rangle \mid v \in V\}$, but the converse is not necessarily true.

EXAMPLE 5.2:

Figure 5.2(a) shows an example of a function and its corresponding pseudoforest. The pseudoforest shown in Fig. 5.2(b) does not correspond to a function. □

We have already introduced *rooted directed trees* (which we call simply *trees*, if the rest is clear from context) in Section 2.2. In particular, if each vertex is directly connected to the root r, then the corresponding directed tree is called a **rooted star.** A rooted directed tree and a rooted star are shown in Fig. 5.3.

A pseudoforest determined by a function consists of a set of rooted directed trees, each with an extra arc from the root to one of the latter's descendants. *When there is no ambiguity, we shall refer to these structures as directed trees or rooted stars.* Notice that a directed tree defined by a function does not have a unique root, since any vertex on the unique cycle determined by the function can be used as a root.



$D(1) = 7, D(2) = 7, D(3) = D(4) = D(5) = 8,$
$D(7) = 10, D(8) = 10, D(10) = 12, D(12) = 7,$
$D(6) = 9, D(9) = 11, D(11) = 13, D(13) = 11$

(a)

$D(9) = ?$

(b)

FIGURE 5.2
Two examples of a pseudoforest. (a) A pseudoforest that corresponds to the specified D function, and (b) a pseudoforest that does not correspond to a function defined on its set of vertices.

The connected-components algorithms generate an output vector D of length n such that $D(u)$ is equal to a representative of the connected component containing u. For example, the smallest vertex of a connected component is a possible choice for a representative of that component. Once the D function is generated, we can answer queries of the form, "Are u and v in the same connected component?" in $O(1)$ sequential time. We can also generate from D efficiently the list of the vertices of each connected component (see Exercise 5.2).

A general strategy to identify the connected components of a graph consists of an iterative procedure, where, at the beginning of each iteration, the available vertices are partitioned into groups such that all the vertices in a group belong to the same connected component. During each iteration, some groups with adjacent vertices are merged into larger groups. The algorithm terminates when no additional groups can be merged. Each group is typically represented by a rooted directed tree, with the root being the representative of that group. The two algorithms differ in the types of directed trees allowed at the end of each iteration, and in which groups are chosen to be merged.

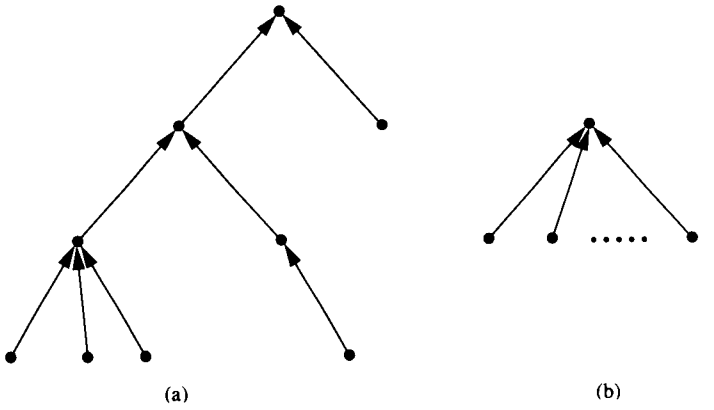


FIGURE 5.3
Two examples of directed trees. (a) A rooted directed tree, and (b) a rooted star.

We begin with the connected-components algorithm that is suitable for handling dense graphs.

5.1.2 AN OPTIMAL ALGORITHM FOR DENSE GRAPHS

Let A be the $n \times n$ adjacency matrix of an undirected graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$. Hence, $A(i,j) = 1$ if and only if $(i,j) \in E$. Define the following function C on V: $C(v) = \min\{u \mid A(u,v) = 1\}$, and, if v is an isolated vertex, then $C(v) = v$. That is, $C(v)$ is the smallest vertex adjacent to v, whenever v is not an isolated vertex; otherwise, $C(v)$ is equal to the vertex v itself.

Lemma 5.1: *Let $G = (V, E)$ be an undirected graph, and let C be the function defined previously. Then, C defines a pseudoforest $\mathcal{F}$ that partitions V into V_1, $V_2, \ldots, V_s$, where each V_i is the set of vertices in a directed tree T_i of $\mathcal{F}$. Moreover, the following claims hold:*

1. *All the vertices in each V_i belong to the same connected component.*
2. *Each cycle in $\mathcal{F}$ either is a self-loop or contains exactly two arcs.*
3. *The cycle of each tree T_i in $\mathcal{F}$ contains the smallest vertex in T_i.*

Proof: We start by proving claim 1. If V_i consists of a single vertex, the claim is vacuously true. Hence, we assume that $|V_i| \geq 2$, and let u and v be two distinct vertices of V_i. Then, $\langle u, C(u)\rangle$ and $\langle v, C(v)\rangle$ belong to the directed tree

T_i. But each arc in T_i corresponds to an edge in G. Hence, there must exist a path connecting u and v in G.

We now establish claims 2 and 3. Note that T_i is a self-loop consisting of the arc $\langle v, v \rangle$ if and only if v is an isolated vertex of G. Suppose that T_i is not a self-loop, and let r be the minimum vertex in T_i. Let $u = C(r)$. Since (u, r) is an edge of the given graph, $C(u)$ must be equal to r. But T_i has exactly one loop, and hence this loop consists of the two arcs $\langle u, r \rangle$ and $\langle r, u \rangle$. Clearly, this loop contains the smallest vertex r in T_i. □

The **root** of a tree T_i is defined to be the smallest vertex in T_i.

Our algorithm begins by computing the C function defined on the set V. As we have shown in Lemma 5.1, the vertices belonging to the same directed tree of the pseudoforest $\mathcal{F}$ induced by C are in the same connected component. Hence, we can "shrink" each set V_i into a single vertex, called a **supervertex,** and repeat the procedure on the supervertex graph defined by the set of supervertices $\{V_i\}$ such that (V_i, V_j) is an edge if and only if there exist $v \in V_i$ and $w \in V_j$ with the property that $(v, w) \in E$. The algorithm terminates when each supervertex becomes isolated.

We must elaborate on two important issues. First, we must clarify what we mean by "shrinking" a set V_i of vertices, and how we obtain the new graph of supervertices. Second, we must specify how to assign, for each vertex $v \in V$, a value $D(v)$ that identifies the vertex's connected component. These two issues are addressed next.

To shrink the set V_i of vertices of a directed tree T_i, we (1) identify a special vertex r_i of V_i and designate it as a representative, and (2) label each vertex of V_i that r_i is its representative. Since T_i is a directed tree, we can apply the pointer jumping technique (Section 2.2) $\lceil \log n_i \rceil$ using the function C, where $n_i = |V_i|$. For each $v \in V_i$, the new value $C(v)$ is equal to one of the two vertices in the loop of T_i. Taking the minimum of $C(v)$ and $C(C(v))$ sets $C(v)$ to the root of its tree, and the modified C function defines now a set of rooted stars. The root r_i of T_i will be chosen as the representative of the set V_i, and will be referred to as the supervertex r_i.

Let r_i and r_j be the two supervertices representing the trees T_i and T_j, respectively. Then, r_i and r_j are adjacent in the supervertex graph if and only if there exist $v \in V_i$ and $w \in V_j$ such that $(v, w) \in E$. We can insert an edge (r_i, r_j) as follows. For each $(x, y) \in E$ such that $C(x) \neq C(y)$, we set $(C(x), C(y))$ to be an edge in the supervertex graph. Notice that a concurrent write of the same value is required to implement this last step.

The second issue of determining, for each vertex v, a value $D(v)$ that identifies the connected component containing v can be handled as follows. When the algorithm terminates, each supervertex r is isolated; hence, we set

$D(r) = r$. These supervertices were the roots of a number of directed trees formed in the previous iteration. We now proceed in the reverse order, by expanding each root r into its tree of the previous iteration, and assigning the value $D(r)$ to all the vertices in that tree. We continue expanding each supervertex until the original graph has been recovered. Since the root of each directed tree is the smallest vertex in that tree, the assigned D value of each vertex v is the minimum vertex in the connected component containing v.

We next present the corresponding overall algorithm.

ALGORITHM 5.1

(Connected Components for Dense Graphs)

Input: *The $n \times n$ adjacency matrix A of an undirected graph.*

Output: *An array D of size n such that $D(i)$ is equal to the smallest vertex in the connected component of i.*

begin

1. Set $A_0 := A, n_0 := n, k := 0$.

2. **while** $n_k > 0$ **do**

2.1. Set $k := k + 1$.

2.2. Set $C(v) := Min\{u \mid A_{k-1}(u, v) = 1, u \neq v\}$ **if** *none* **then** v.

2.3. Shrink each tree of the pseudoforest defined by C to a rooted star. The root of each star containing more than one vertex defines a new supervertex.

2.4. Set n_k to be equal to the number of the new supervertices, and set A_k to be the $n_k \times n_k$ adjacency matrix of the new supervertex graph.

3. For each vertex v, determine $D(v)$ as follows. If, at the end of step 2, $C(v) = v$, then set $D(v) = v$; otherwise, reverse the process performed at step 2 by expanding each supervertex r (of the supervertices formed during an iteration) into the set V_r of vertices of its directed tree, and making the assignment $D(v) = D(r)$ for each $v \in V_r$.

end

Remark 5.1: During the kth iteration of Algorithm 5.1, we can assign to the new supervertices $\{r_i\}_{i=1}^{n_k}$ serial numbers from $\{1, 2, \cdots, n_k\}$ by using the prefix-sums algorithm (Section 2.1); the numbers are then used in determining the adjacency matrix A_k. The information regarding the serial numbers should be recorded somewhere so that, during the execution of step 3, each vertex can be assigned the correct D value. Step 3 expands the supervertices in reverse order to that in which they were merged in step 2. This process is

similar to that used in the contraction procedure for the optimal list-ranking algorithm (Section 3.1). The reader is asked to provide the details in Exercise 5.3. □

EXAMPLE 5.3:

Consider the graph shown in Fig. 5.4(a). During the first iteration of the loop, the C function defined at step 2.2 is given by $C(1) = 5, C(5) = 1, C(4) = 2, C(2) = 3, C(3) = 2, C(6) = 3, C(7) = 6, C(8) = 9$, and $C(9) = 8$. Hence, the corresponding pseudoforest has three directed trees, as shown in Fig. 5.4(b). Step 2.3 causes each of these rooted trees to become a rooted star, as shown in Fig. 5.4(c). Assign to the roots 1, 2, and 8 the serial numbers $s(1) = 1, s(2) = 2$, and $s(8) = 3$. The adjacency matrix A_1 is given by

$$A_1 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

During the second iteration, the function C yields $C(1) = 2, C(2) = 1$, and $C(3) = 3$; hence, we have one nontrivial star rooted at 1. In this case, $n_2 = 1$, and the nontrivial star is changed into a self-loop during the third iteration; hence, $n_3 = 0$. Therefore, the algorithm terminates with two stars.

We conclude that there are two connected components: one whose representative is vertex 1, the other whose representative is vertex 8 (which was assigned the serial number of 3 during the first iteration). Hence, at step 3, $D(1) = 1$ and $D(8) = 8$. We now expand the supervertices in the reverse order. Vertex 2 belongs to the star with root 1 obtained at the second iteration. Hence, $D(2) = D(1) = 1$. The remaining vertices belonged to stars during the first iteration. Hence, we obtain $D(5) = D(1) = 1, D(3) = D(4) = D(6) = D(7) = D(2) = 1$, and $D(9) = D(8) = 8$. □

Lemma 5.2: *Let n_k be the number of nontrivial stars at the end of the kth iteration of step 2 in Algorithm 5.1. Then, $n_k \leq n_{k-1}/2$, for all $k \geq 1$.*

Proof: At the end of the kth iteration, some of the n_{k-1} stars determined by the $(k-1)$st iteration will become self-loops (and hence *trivial*)—say, n'_{k-1} such stars—and each of the remaining $n_{k-1} - n'_{k-1}$ stars will be contained in one of the n_k nontrivial stars. Thus, $n_k \leq (n_{k-1} - n'_{k-1})/2$, since each nontrivial star contains at least two vertices. Therefore, $n_k \leq n_{k-1}/2$. □

We now establish the following theorem.

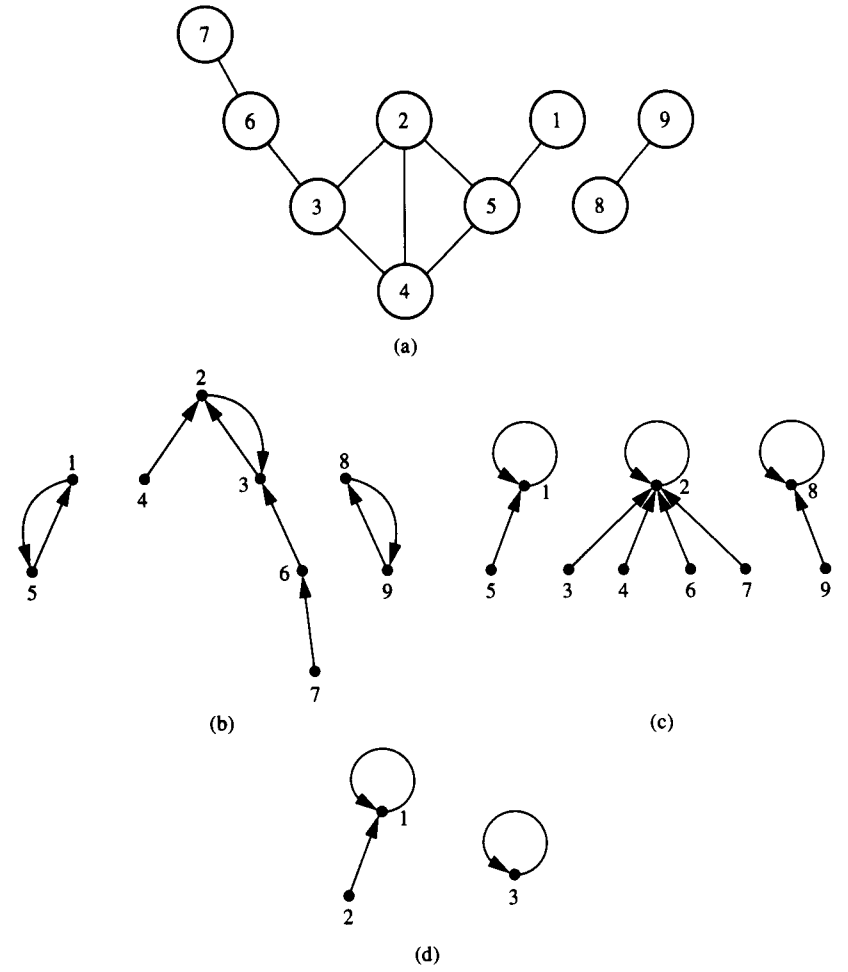


FIGURE 5.4
Results of Algorithm 5.1 used on a graph, as explained in Example 5.3. (a) An input graph to the connected components for dense-graphs algorithm. (b) The pseudoforest generated by the C function that assigns to each vertex v the smallest vertex adjacent to v. (c) The shrinking of the directed trees. (d) The pseudoforest obtained during the second iteration, where the supervertices 1, 2, and 3 stand for the three rooted stars in (c) whose roots are vertices 1, 2, and 8 respectively.

Theorem 5.1: *Given the $n \times n$ adjacency matrix of an undirected graph G, Algorithm 5.1 determines the connected components of G in $O(\log^2 n)$ time, using a total of $O(n^2)$ operations.*

Proof: Let n be the number of vertices in the input graph G. We prove by induction on n that, at the end of the algorithm, $D(v)$ will be equal to the smallest vertex in the connected component containing v.

The base case $n = 1$ corresponds to a single vertex v in G. The first iteration will make v into a supervertex, since $C(v) = v$. There are no nontrivial stars in this case; hence, we go immediately to step 3, where we obtain that $D(v) = v$.

Suppose that $n > 1$. Let v be an arbitrary vertex of G. If v is an isolated vertex, then it is easy to see that v will be a supervertex by itself until the termination of the algorithm. Hence, there is nothing to prove in this case.

Assume that v is not an isolated vertex of G. During the first iteration, v will belong to a rooted star (step 2.3) whose root r is the smallest vertex in the star. If $v = r$, then v will be one of the supervertices in the new supervertex graph—say, G'. By the induction hypothesis, $D(v)$ will be assigned the smallest vertex in its connected component in G'. The proof follows when we observe that each of the supervertices is the smallest vertex of its star. The remaining case is when $v \neq r$; then, $D(v)$ will be set equal to $D(r)$ at step 3, and the proof follows as before.

We now present the analysis of the resources required by the algorithm.

Consider the kth iteration of the **while** loop of step 2. Determination of C (step 2.2) can be performed easily in $O(\log n_{k-1})$ time, using $O(n_{k-1}^2)$ operations, since it involves the computation of n_{k-1} minima, each involving a row containing n_{k-1} elements. We can shrink each tree defined by C to a star (step 2.3) by using the pointer jumping technique (see Section 2.2 and Exercise 2.41). If we apply the pointer jumping technique for $\lceil \log n_{k-1} \rceil$ iterations, each vertex will be pointing to one of the two vertices making up the loop of its tree. Taking the minimum of these two vertices will transform each tree into a rooted star with a self-loop at the root. Hence, step 2.3 takes $O(\log n_{k-1})$ time, using $O(n_{k-1} \log n_{k-1})$ operations. Determining the number n_k of the nontrivial supervertices is a prefix-sums operation that can be performed in $O(\log n_k)$ time, using $O(n_k)$ operations. The adjacency matrix A_k can be determined as follows. For each $A_{k-1}(u, v) = 1$, set $A_k(r(u), r(v)) = 1$, where $r(x)$ is the serial number of $C(x)$ (which is the root of the star containing x). This step takes $O(1)$ time, using $O(n_{k-1}^2)$ operations, if we allow concurrent write of the same value instructions.

It follows that each iteration of the **while** loop takes $O(\log n_{k-1})$ time, using a total of $O(n_{k-1}^2)$ operations.

The number of iterations is $O(\log n)$, since $n_k \leq n_{k-1}/2$ (by Lemma 5.2) and $n_0 = n$. The total number of operations required for the completion of the **while** loop is $\sum_k O(n_k^2) = \sum_k O(n^2/2^k) = O(n^2)$.

Step 3 depends on how the information concerning the rooted stars and their serial numbers has been recorded. It is not difficult to confirm that this step can be performed within the bounds stated for step 2 (the details are left to Exercise 5.3). □

Notice that Algorithm 5.1 is *optimal* whenever the adjacency matrix is given as input or if $|E| = \Theta(n^2)$.

PRAM Model: Steps 1, 2.1, and 2.2 of Algorithm 5.1 do not require any simultaneous memory access. Step 2.3 requires concurrent-read capability, and our method for constructing A_k in step 2.4 requires concurrent write of the same value. Step 3 can be implemented without any simultaneous memory access. Hence, Algorithm 5.1 can be implemented on the common CRCW PRAM model within the stated bounds. Adaptation of this algorithm to the CREW PRAM model is left to Exercise 5.1. □

Remark 5.2: Step 2.2 selects, for each v, the smallest vertex adjacent to v. Actually, the algorithm will work correctly if an *arbitrary* adjacent vertex is selected. The shrinking at step 2.3 can still be done efficiently. The reader is asked to provide the details of such an algorithm and its corresponding complexity bounds in Exercise 5.5. □

5.1.3 AN EFFICIENT ALGORITHM FOR SPARSE GRAPHS

Each iteration of Algorithm 5.1 requires $O(\log n)$ time, essentially because we insist on forming rooted stars at the end of each iteration. In addition, the definition of the function C is unecessarily restrictive. The next algorithm will relax these two restrictions such that each iteration can be executed in $O(1)$ time.

As in the previous connected-components algorithm (Algorithm 5.1), we manipulate a forest of directed trees such that the vertices in each tree belong to the same connected component. In this case, each directed tree is constructed with a self-loop at its root. During each iteration, we examine each edge (u, v) of the input graph, and, under certain conditions, we combine the two trees containing u and v if they are distinct. At the same time, we shorten the heights of the trees by applying the pointer jumping technique to each vertex in a directed tree.

Let D be the function on V defining a pseudoforest that arises at the beginning of an iteration. Initially, we set $D(v) = v$, for each $v \in V$. There are two basic operations performed on the pseudoforest during each iteration, which we define as follows:

- **Grafting:** Let T_i and T_j be two distinct trees in the pseudoforest defined by D. Given the root r_i of T_i and a vertex v of T_j, the operation $D(r_i) = v$ is called *grafting* T_i *onto* T_j (see Fig. 5.5).

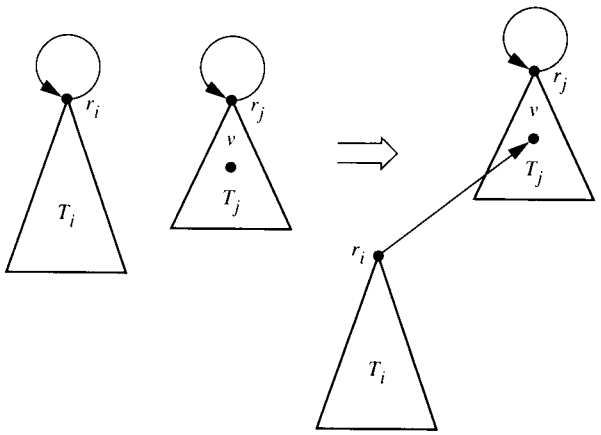


FIGURE 5.5
The operation of grafting tree T_i onto vertex v of T_j. The two directed trees on the left side are combined to form a single rooted tree shown on the right side.

- **Pointer jumping:** Given a vertex v in a tree T, the *pointer jumping* operation applied to v is to set $D(v) = D(D(v))$ (see Fig. 5.6).

We now elaborate on the applications of these two operations during each iteration of our new connected-components algorithm.

Let T_i and T_j be two directed trees of our pseudoforest at the beginning of an iteration. Suppose that there exists an edge $(u, v) \in E$ such that $u \in T_i$ and $v \in T_j$. We clearly should try to merge the two trees T_i and T_j, since their vertices belong to the same connected component. To accomplish such a merging step in $O(1)$ time, we require that either u or v be a root or directly connected to a root of its tree. A vertex x satisfies the latter condition if and only if $D(x) = D(D(x))$; hence, such a condition can be verified in $O(1)$ time. However, a problem arises if both u and v satisfy this condition—that is, if $D(u) = D(D(u))$ and $D(v) = D(D(v))$—which will result in an attempt to graft T_i onto T_j and T_j onto T_i simultaneously. To avoid such a difficulty, we insist on grafting a tree onto a *smaller* vertex of another tree. That is, given an edge $(u, v) \in E$, we check whether $D(u) = D(D(u))$ *and* $D(v) < D(u)$, and, if they are, we graft T_i onto T_j—that is, we set $D(D(u)) = D(v)$.

Grafting a tree into a smaller vertex of another tree may in itself create a difficulty of a different type. Consider the case when the tree T_i is a rooted star such that all of its vertices are smaller than any of their adjacent vertices (Fig. 5.7). In this case, T_i may not be merged with any other tree until (1) one

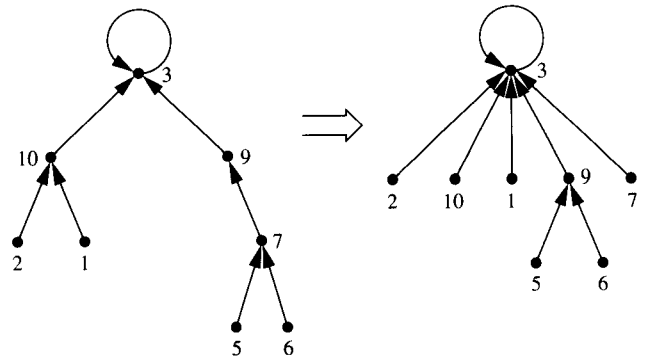


FIGURE 5.6
The pointer jumping operation applied to each vertex of the directed tree on the left side. The resulting directed tree is shown on the right side.

of the adjacent vertices becomes directly connected to the root of its tree or becomes the root, and (2) that tree does not get grafted onto a tree different from T_i (see Fig. 5.7). To avoid this problem, we require that, after the initial grafting process has been completed, an attempt be made to graft a star with root r_i onto the vertex v, whenever there exists an edge connecting the star T_i and v.

The pointer jumping operation is applied to each vertex v. Hence, the height of the directed tree containing v reduces by at least a factor of 2/3 whenever the tree is not a rooted star.

We are ready to present the algorithm. The input consists of a set of edges (i,j) given in an arbitrary order. An edge (i,j) will appear twice, as (i,j) and as (j,i). In addition, for each vertex i, we introduce a dummy vertex i' connected to i. We initialize the function D with $D(i) = D(i') = i$, for all i. The reason for introducing the dummy vertices i' is to avoid the possibility of trying to graft two stars onto each other during the first iteration of the algorithm. Hence, we are starting with n stars.

The algorithm is based on an iterative procedure specified as follows.

ALGORITHM 5.2

(Connected Components for Sparse Graphs)

Input: *(1) A set of edges (i,j) given in an arbitrary order, and (2) a pseudoforest defined by a function D such that all the vertices in each tree belong to the same connected component.*

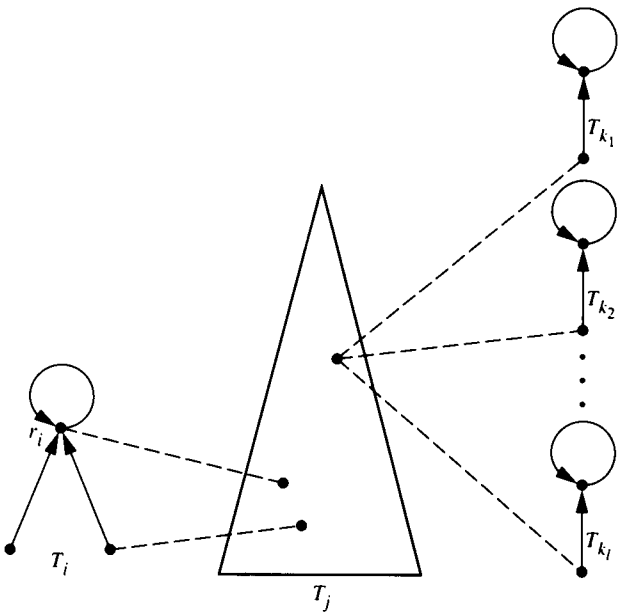


FIGURE 5.7
The main problem of grafting a tree onto a smaller vertex. The vertices of star T_i are smaller than all of the adjacent vertices in T_j. The directed tree T_j has also smaller adjacent vertices in the trees $T_{k_1}, \ldots, T_{k_l}$, such that there are no edges between any two T_{k_j}'s. During an iteration, T_j may get grafted onto any of $T_{k_1}, \ldots, T_{k_l}$, and T_i. Hence, T_j can get grafted consecutively to $T_{k_1}, \ldots, T_{k_l}$, and finally to T_i.

Output: *The pseudoforest obtained after (1) grafting trees onto smaller vertices of other trees, (2) grafting rooted stars onto other trees if possible, and (3) performing the pointer jumping operation on each vertex.*

Description of a general iteration

begin

1. Perform a grafting operation of trees onto smaller vertices of other trees as follows:

for *all* $(i, j) \in E$ **pardo**

if $(D(i) = D(D(i))$ **and** $D(j) < D(i))$ **then** *set* $D(D(i)) := D(j)$

2. Graft rooted stars onto other trees if possible, as follows:

for *all* $(i, j) \in E$ **pardo**

if (i belongs to a star **and** $D(j) \neq D(i)$) **then** set $D(D(i)) := D(j)$

3. If all the vertices are in rooted stars, then **exit**. *Otherwise, perform the pointer jumping operation on each vertex as follows:*
for *all i* **pardo**
Set $D(i)$: $= D(D(i))$
end

Remark 5.3: Each of steps 1 and 2 in Algorithm 5.2 may attempt to graft a single tree onto several other trees. In this case, we assume that one of them succeeds, and we do not care which one. The arbitrary CRCW PRAM is suitable for implementing such a policy. □

Remark 5.4: To test whether a vertex i belongs to a star, we can do the following:

for all vertices i **pardo**
Set $star(i)$: $=$ **true**
if $D(i) \neq D(D(i))$ **then**
Set $star(i), star(D(i)), star(D(D(i)))$: $=$ **false**
Set $star(i)$: $= star(D(i))$

At the termination of this procedure, $star(i) =$ **true** if and only if i belongs to a star. This procedure can be implemented in $O(1)$ time, using a linear number of operations. □

EXAMPLE 5.4:

Let G be the graph given in Fig. 5.8(a). Initially, $D(i) = i$ and each vertex is in a rooted star, as shown in Fig. 5.8(b). At step 1 of the first iteration, several grafting operations of the same tree are attempted. For example, the tree rooted at 6 could be grafted onto the vertex 1 (because of edge (6, 1)) or onto vertex 5 (because of edge (6, 5)). We assume, in this case, that vertex 6 is grafted onto vertex 1. Fig. 5.8(c) shows the results of the various grafting operations performed at step 1 during the first iteration. Step 2 grafts the star rooted at vertex 4 onto the tree rooted at vertex 2, as shown in Fig. 5.8(d). Step 3 reduces the height of each tree, as shown in Fig. 5.8(e). After step 1 of the second iteration has been executed, we obtain the pseudoforest of Fig. 5.8(f). Step 2 has no effect during the second iteration. At the end of the second iteration, the vertices of each connected component belong to the same directed tree. The next two iterations reduce the two trees into rooted stars, and the algorithm terminates. □

Remark 5.5: If step 2 of Algorithm 5.2 is omitted, then it may take $\Omega(n)$ iterations for the algorithm to terminate. Providing such an example is left to Exercise 5.8. □

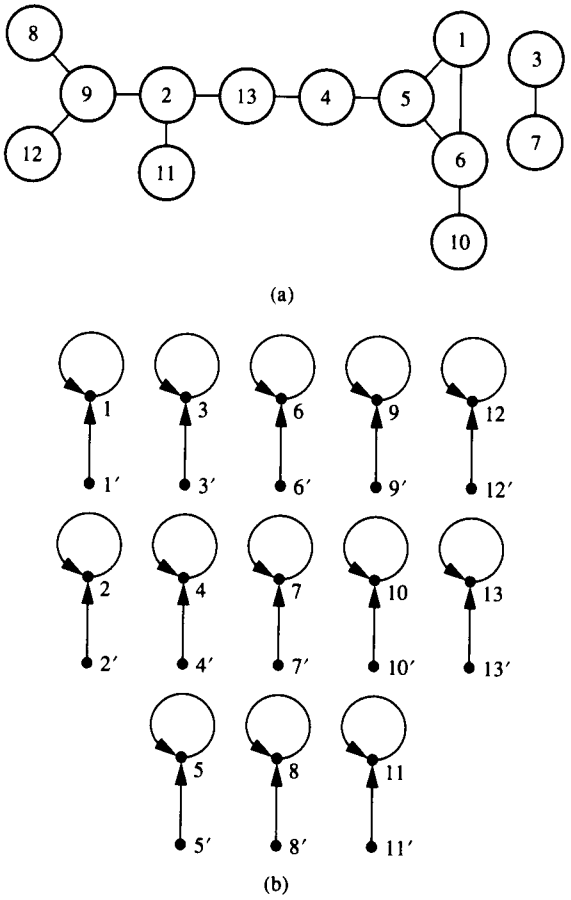


FIGURE 5.8
Illustration of the connected-components algorithm for sparse graphs (Algorithm 5.2). (a) The initial graph. (b) The rooted stars prior to the first general iteration.

Theorem 5.2: *When Algorithm 5.2 terminates, all the vertices in the same connected component of a root r are directly connected to r. All the trees become rooted stars after* $O(\log n)$ *iterations, each iteration requiring* $O(1)$ *time. The total number of operations required by the algorithm is* $O((m + n) \log n)$.

Proof: We shall prove claims 1 and 2, which establish the correctness of Algorithm 5.2.

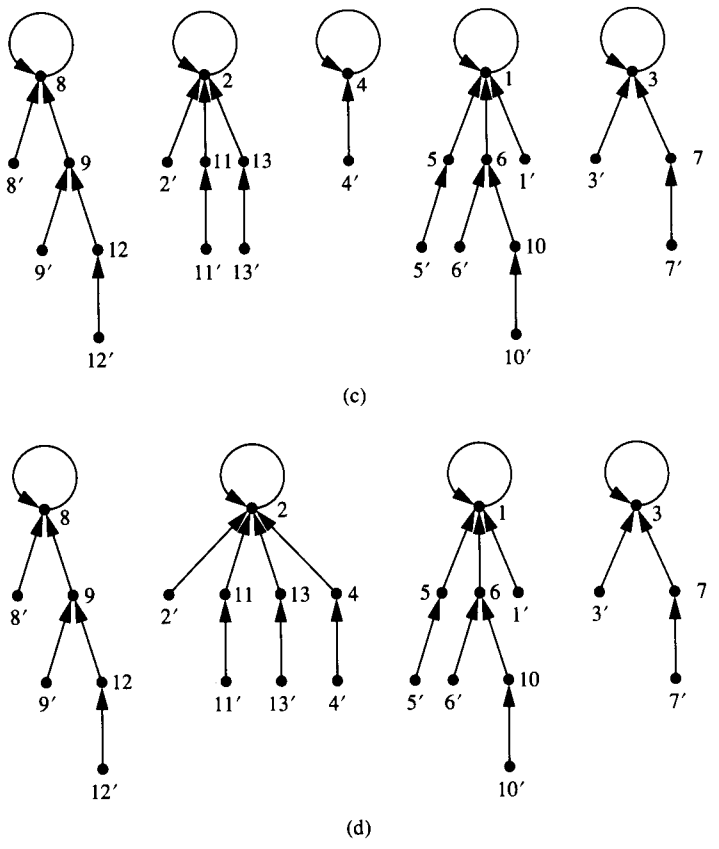


FIGURE 5.8 (*continued*)
(c) The directed trees generated by the grafting operations during the first iteration. (d) The result of grafting the star rooted at vertex 4 onto the tree rooted at vertex 2. *Figure 5.8 continues on page 220.*

Claim1: At the end of each iteration, the pseudoforest defined by D consists of directed trees with self-loops at their roots such that all vertices in a tree belong to the same connected component.

Claim2: In Algorithm 5.2, let r be the root of a star after step 2 of some iteration. All the vertices v in the connected component of r satisfy $D(v) = r$, that is, they belong to the star rooted at r.

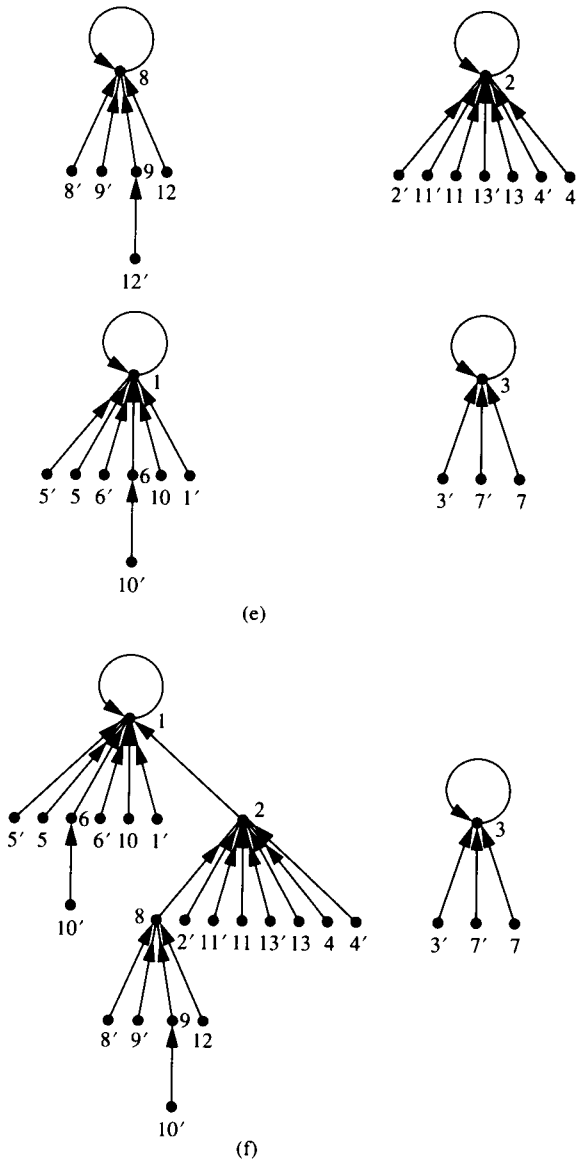


FIGURE 5.8 (*continued*)
(e) The directed trees generated at the end of the first iteration. (f) The directed trees after step 1 of the second iteration.

Proof of Claim1: The proof is by induction on the iteration number k. The base case $k = 0$ follows immediately, since each vertex is in a rooted star.

Assume that the induction hypothesis holds for the kth iteration. During the $(k + 1)$st iteration, a grafting operation performed at step 1 or step 2 will combine two trees whose vertices belong to the same connected component. The effect of step 3 is to shorten the heights of the directed trees that are not stars. In each case, the root of each resulting tree will keep its self-loop. Therefore, the induction hypothesis holds for all values of k, and Claim1 follows.

Proof of Claim2: Suppose that there exists a vertex v in the connected component of r such that $D(v) \neq r$ (the D value in the current iteration). The existence of this vertex implies the existence of a vertex w such that $D(w) \neq r$, and w is connected with an edge to the tree containing r. But then r would have been grafted onto another tree at step 2. Therefore, all the vertices u in the connected component of r satisfy $D(u) = r$.

To establish that the algorithm terminates after $O(\log n)$ iterations, we show the following claim.

Claim3: Let K be a connected component of G and let $|K|$ denote the number of vertices in K. Let $h_k(K)$ be the sum of the heights of the trees consisting of the vertices of K at the end of the kth iteration. If these trees do not form a single rooted star, then $h_k(K) \leq \left(\frac{2}{3}\right)^k |K|$.

Proof of Claim3: The proof is by induction on k. The base case $k = 0$ holds since we always have $h_k(K) \leq |K|$.

Denote by $h(T)$ the height of a tree T. Since no tree is ever grafted onto a leaf, grafting a tree T_i onto another tree T_j yields a tree T whose height satisfies $h(T) \leq h(T_i) + h(T_j)$. Hence, after steps 1 and 2, $h_k(K)$ does not increase. Moreover, if these trees are not transformed into a single star, none of them is a rooted star (because of step 2). By the pointer jumping operation of step 3, $h_k(K) \leq \left(\frac{2}{3}\right) h_{k-1}(K)$. The proof follows by induction.

Using Claim3, it is clear that $h(T) \leq 1$ for each tree T after $O(\log n)$ iterations. Each iteration requires $O(m + n)$ operations, and hence the total number of operations is $O((m + n) \log n)$. □

PRAM Model: Step 3 of Algorithm 5.2 requires concurrent-read capability only. In each of steps 1 and 2, the algorithm may try to graft a given tree to several trees. In this case, a concurrent write will take place, and we assume that any one of them succeeds. Hence, our algorithm runs on the arbitrary CRCW PRAM model. □

5.2 Minimum Spanning Trees

Let $G = (V, E)$ be a connected, undirected graph with a weight function w on the set E of edges to the set of reals. A **spanning tree** is a subgraph $T = (V, E_T)$, $E_T \subseteq E$, of G such that T is a tree. The weight $w(T)$ of a spanning tree T is the sum of the weights of its edges. A spanning tree with the smallest possible weight is called a **minimum spanning tree (MST)** of G. Determining an MST of a given weighted graph is an important problem that arises in many applications.

Assume without loss of generality that no two edges have the same weight. Otherwise, we can assign, for each edge e, a new weight $w'(e)$ consisting of the pair $(w(e), s(e))$, where $w(e)$ is the given edge weight, and $s(e)$ is its serial number. Given any two distinct edges e_1 and e_2, we have that $w'(e_1) \neq w'(e_2)$ because $s(e_1) \neq s(e_2)$; in addition, $w'(e_1) < w'(e_2)$ if $w(e_1) < w(e_2)$, or $w(e_1) = w(e_2)$ and $s(e_1) < s(e_2)$. The sum of the weights of several edges is computed componentwise.

5.2.1 A STRATEGY FOR COMPUTING MINIMUM SPANNING TREES

The following observation leads to an efficient strategy for solving the MST problem.

Lemma 5.3: *Let $G = (V, E)$ be a weighted connected graph. For each vertex $u \in V$, let $C(u) \in V$ be such that $(u, C(u))$ is the minimum-weight edge incident on u. Then, the following two claims hold:*

1. *All the edges $(u, C(u))$ belong to the MST.*
2. *The function C defines a pseudoforest such that each directed tree has a cycle containing exactly two arcs.*

Proof: We start with the proof of Claim1. Let T be a minimum spanning tree. Suppose that $(u, C(u))$ is not in T for some $u \in V$. Let $P = (u, x_1, \ldots, x_s, C(u))$ be the path in T connecting u to $C(u)$. Replacing (u, x_1) with $(u, C(u))$ yields another spanning tree T' such that $w(T') < w(T)$. This result contradicts the fact that T is an MST; hence, $(u, C(u))$ must be in T.

As for Claim2, note that, if a directed tree T' defined by C contains a cycle of length greater than 2, the undirected version of T' will contain a cycle. This result is impossible since, by Claim1, all these edges must belong to the MST. Moreover, no self-loops can occur, since $(u, C(u))$ must be an edge of G for each u. Therefore, each directed tree must have a cycle containing exactly two arcs. □

The proof of Lemma 5.3 can be slightly generalized to obtain the following lemma.

Lemma 5.4: *Let $V = \cup V_i$ be an arbitrary partition of V with the corresponding subgraphs $G_i = (V_i, E_i)$, where $1 \leq i \leq t$. For each i, let e_i be the minimum-weight edge connecting a vertex in V_i to a vertex in $V - V_i$. Then, all the edges e_i belong to an MST of the graph $G = (V, E)$.* □

Note that, since all the edges have different weights, Lemma 5.4 implies that *the MST is unique*.

There are three well-known greedy strategies to solve the MST problem, all of which can be derived from Lemmas 5.3 and 5.4. The first, which leads to **Prim's algorithm,** grows a tree successively by adding minimum-weight edges to it, starting initially from a single vertex. The second strategy, leading to **Kruskal's algorithm,** processes the edges in order of their weights, adding those that do not create cycles. The third strategy leads to **Sollin's algorithm** and is described in Section 5.2.2.

We note that there are faster sequential algorithms to solve the MST problem on sparse graphs, but these algorithms are more involved than is any of the three algorithms sketched here. The interested reader is referred to the bibliographic notes at the end of this chapter.

5.2.2 SOLLIN'S ALGORITHM AND ITS PARALLEL IMPLEMENTATION

Sollin's MST algorithm begins with the forest $F_0 = (V, \emptyset)$ and grows trees on subsets of V until there is a single tree containing all the vertices. During each iteration, the minimum-weight edge incident on each tree is selected. The new edges are added to the current forest—say, F_s—to obtain a new forest, F_{s+1}. This process is continued until there is only a single tree. It is clear that the number of trees in F_{s+1} is at most one-half the number of trees in F_s. Hence, Sollin's algorithm requires $O(\log n)$ iterations.

Our parallel implementation of Sollin's algorithm runs in $O(\log^2 n)$ time, using a total of $O(n^2)$ operations. An alternative implementation with running time $O(\log n)$ and a total of $O(m \log n)$ operations is left to Exercise 5.13.

Let W be the weight matrix of the given connected graph $G = (V, E)$. For $i \neq j$, $W(i, j)$ is equal to the weight of the edge (i, j), if the latter exists; otherwise, $W(i, j) = \infty$, where ∞ is an identity element with respect to the minimum operator. We also set $W(i, i) = \infty$.

When the algorithm terminates, each edge belonging to the MST is marked. We can then generate efficiently the list of the edges in MST.

The strategy of connected-components algorithm for dense graphs (Algorithm 5.1) is used to implement Sollin's algorithm. The only essential difference is the definition of the function C. In the next algorithm, C will be used to select the edges in the MST.

ALGORITHM 5.3
(Minimum Spanning Tree)

Input: *An $n \times n$ array W, representing a weighted connected graph such that no two edges have the same weight.*
Output: *A label for each edge belonging to the MST.*

begin
1. *Set* $W_0 := W, n_0 := n, k := 0$
2. **while** $(n_k > 1)$ **do**
   - *2.1. Set* $k := k + 1$
   - *2.2. Set* $C(v) := u$, *where* $W_{k-1}(v, u) = \min\{W_{k-1}(v, x) \mid x \neq v\}$. *Mark* $(v, C(v))$.
   - *2.3. Shrink each directed tree of the pseudoforest defined by C to a rooted star.*
   - *2.4. Set n_k to be equal to the number of the rooted stars, and set W_k to be the $n_k \times n_k$ weight matrix of the graph induced by viewing each star as a single vertex.*
3. *Restore each marked edge to its original name.*

end

Remark 5.6: Note that if r is a root of a tree in the pseudoforest defined by C at step 2.2, then the edge $(r, C(r))$ will get marked "twice." The handling of this problem depends on the exact form of the output desired. For example, we can output the adjacency matrix of the MST, in which case $(r, C(r))$ and $(C(r), r)$ correspond to two different entries; we can then make the corresponding matrix symmetric at the very end of the algorithm. This solution will not increase the resource requirements of the algorithm. □

EXAMPLE 5.5:

Consider the weighted graph G given in Fig. 5.9(a). The C function determined during the first iteration of the **while** loop is given by $C(v_1) = v_6, C(v_2) = v_3, C(v_3) = v_4, C(v_4) = v_3, C(v_5) = v_3$, and $C(v_6) = v_1$ (Fig. 5.9b). The edges (1, 6), (2, 3), (3, 4), and (3, 5) are marked. Step 2.3 shrinks each of the two trees defined by C into rooted stars, one with root v_4, and the other with root v_1 as shown in Fig. 5.9(c). Assign serial numbers 1 and 2 to vertices 4 and 1. The matrix W_1 is then given by

$$W_1 = \begin{bmatrix} \infty & 4 \\ 4 & \infty \end{bmatrix}.$$

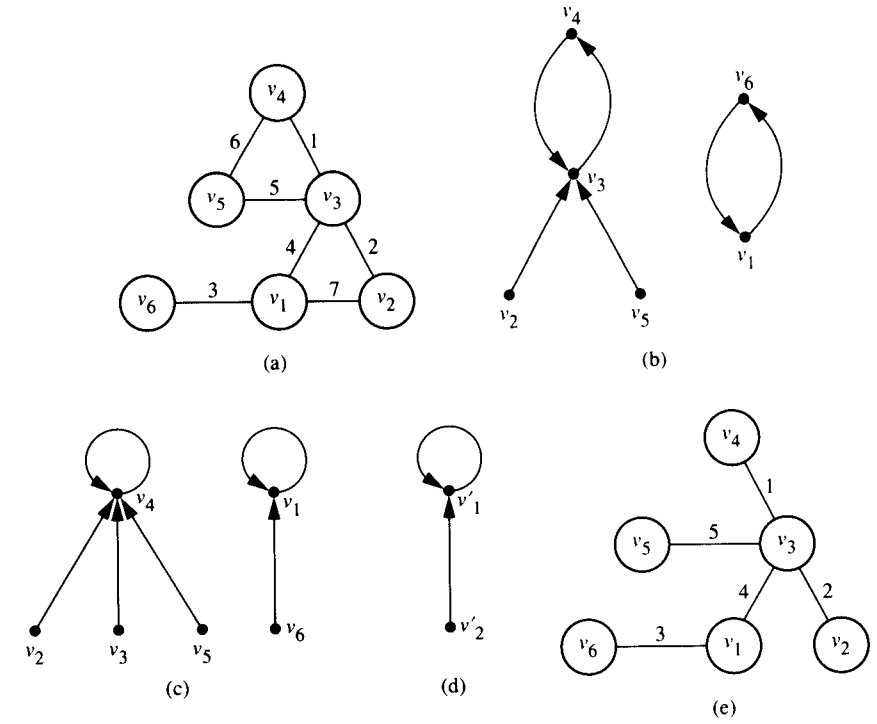


FIGURE 5.9
An illustration of the MST algorithm (Algorithm 5.3). (a) An input graph. (b) The pseudoforest generated during the first iteration. (c) The stars obtained after application of the pointer jumping technique to each directed tree. (d) The pseudoforest obtained during the second iteration, where v'_1 represents the set $\{v_2, v_3, v_4, v_5\}$ and v'_2 represents the set $\{v_1, v_6\}$, and the edge between v'_1 and v'_2 is edge (v_3, v_1), which is the minimum-weight edge connecting the two corresponding sets of vertices. (e) The resulting MST.

Hence, during the second iteration we get $C(1) = 2$ and $C(2) = 1$. This result implies that $n_2 = 1$, and we thus go to step 3. Edge (1, 3) will then be recovered as the marked edge during the second iteration. □

The correctness of Algorithm 5.3 follows from Lemmas 5.3 and 5.4, and from the correctness proof of Algorithm 5.1. The complexity analysis is identical to that of Algorithm 5.1, except for step 2.4. We now outline the implementation details of this step.

Let G_t denote the graph defined by the matrix W_t. After the execution of step 2.3, the vertices in G_{k-1} are grouped into supervertices such that i and j are in the same supervertex if and only if $C(i) = C(j)$. The purpose of step 2.4 is to obtain the $n_k \times n_k$ matrix W_k of the induced graph G_k from the $n_{k-1} \times n_{k-1}$ matrix W_{k-1} of graph G_{k-1}.

Let r_1 and r_2 be the roots of two distinct rooted stars. For simplicity, assume that r_1 and r_2 denote also the serial numbers of the corresponding stars. Then, $W_k(r_1, r_2) = \min\{W_{k-1}(i,j) \mid C(i) = r_1 \text{ and } C(j) = r_2\}$. We can compute the weight matrix W_k in two stages.

- *Stage 1:* Compute the $n_{k-1} \times n_k$ weight matrix W'_{k-1} defined by $W'_{k-1}(i,r) = \min\{W_{k-1}(i,j) \mid C(j) = r\}$, where i is a vertex of G_{k-1} and r is a vertex of G_k.

  Explanation: We group the columns of the matrix W_{k-1} into n_k groups according to their supervertices, which we can accomplish by sorting the vertices of G_{k-1} according to their C values. We can now compute the matrix W'_{k-1} by replacing each row of the modified W_{k-1} by n_k minima; each minimum is taken over a group.

  The sorting step can be performed in $O(\log n_{k-1})$ time, using a total of $O(n_{k-1} \log n_{k-1})$ operations (pipelined merge sort, Algorithm 4.4). Each row of W'_{k-1} requires $O(\log n_{k-1})$ time, using $O(n_{k-1})$ operations. Therefore, stage 1 can be completed in $O(\log n_{k-1})$ time, using a total of $O(n_{k-1}^2)$ operations.

- *Stage 2:* Compute the $n_k \times n_k$ weight matrix such that $W_k(r_1, r_2) = \min\{W_{k-1}(i,j) \mid C(i) = r_1 \text{ and } C(j) = r_2\}$, for $1 \le r_1, r_2 \le n_k$.

  Explanation: The weight matrix W_k can be generated from W'_{k-1} in exactly the same way that W'_{k-1} was generated from W_{k-1}, except that we group the rows according to their supervertices, instead of grouping the columns. Hence, this stage takes also $O(\log n_{k-1})$ time, using a total of $O(n_{k-1}^2)$ operations.

  We therefore have the following theorem.

Theorem 5.3: *Given the $n \times n$ weight matrix of a connected, undirected graph $G = (V, E)$, an MST can be found in $O(\log^2 n)$ time, using a total of $O(n^2)$ operations. Hence Algorithm 5.3 is optimal.*

PRAM Model: Step 2.2 of Algorithm 5.3 does not require any simultaneous memory access; steps 2.3 and 2.4 require concurrent-read capability. Hence, our algorithm is of the CREW PRAM type. □

Remark 5.7: We can solve the problem of determining an **arbitrary spanning tree** of an undirected, connected graph in a similar fashion by adapting

either of the connected-components algorithms presented in Section 5.1 (Algorithms 5.1 and 5.2). Therefore, a spanning tree of a connected graph with n vertices and m edges can be found either (1) in $O(\log^2 n)$ time, using $O(n^2)$ operations on the CREW PRAM, or (2) in $O(\log n)$ time, using $O((m + n) \log n)$ operations on the arbitrary CRCW PRAM with G given as a sequence of edges. □

5.3 Biconnected Components

Let $G = (V,E)$ be a connected, undirected graph, such that $|V| = n$ and $|E| = m$. Define a relation R_b on E as follows. Given any two edges e and g, $eR_b g$ if and only if $e = g$ or e and g are on a common simple cycle. It is straightforward to check that R_b is an equivalence relation, and hence partitions E into equivalence classes $\{E_i\}_{i=1}^{s}$. Let $V_i = \{v \in V \mid v \text{ is the endpoint of some edge in } E_i\}$. Then, the subgraphs $G_i = (V_i, E_i)$ are called the **biconnected components** or **blocks** of G. Note that a vertex may appear in several blocks.

EXAMPLE 5.6:

Let G be the graph given in Fig. 5.10(a). Then, G has four blocks, as shown in Fig. 5.10(b). Each block either is a single edge or consists of several edges such that any pair of these edges lies on a common simple cycle. Note that vertex v_1 belongs to three blocks. □

It is well known that determining the blocks of G can be done in $O(n + m)$ sequential time using depth-first search. Our parallel algorithm follows a different strategy that is based on transforming the graph G into another graph G' such that the blocks of G are the connected components of G'. The starting point is an *arbitrary* spanning tree, rather than a depth-first search tree.

5.3.1 REDUCTION TO CONNECTED COMPONENTS

We begin by reviewing a few standard definitions from graph theory.

Let C_1 and C_2 be two cycles of the input graph $G = (V, E)$. The **exclusive OR** of C_1 and C_2, denoted by $C_1 \oplus C_2$, is the set of edges in C_1 or in C_2, but not in both. Let $\{C_i\}_{i=1}^{s}$ be a set of cycles in G. The cycles C_1, C_2, ... , C_s are called **independent** if there exists no cycle C_i that is the exclusive OR of a subset of the remaining cycles. A collection $\mathcal{C}$ of *independent* cycles is called a **cycle basis** for G if and only if *every* cycle of G is the exclusive OR of a subset of $\mathcal{C}$.

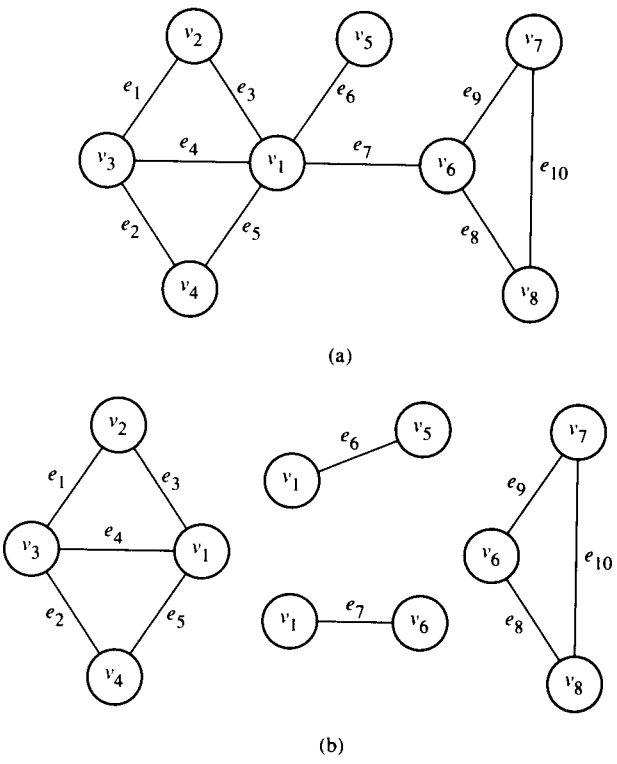


FIGURE 5.10
The graph for Example 5.6. (a) The graph. (b) The graph's blocks; each block either is a single edge or consists of several edges, any two of which lie on a common simple cycle.

It is not clear from these definitions whether every graph has a basis cycle. Each one does, however, as we show next. Assume that G is connected; otherwise, we can treat each of the connected components separately. A spanning tree $T = (V, E_T)$ of the graph G determines a cycle basis as follows. Each edge $e \in E - E_T$ creates a unique cycle C_e when added to T. The collection $\{C_e \mid e \in E - E_T\}$ forms a cycle basis for G. Each such cycle C_e will be called a **basis cycle** relative to T. Since we are assuming that G is connected, the number of elements in the cycle basis is equal to $|E| - |E_T| = m - n + 1$.

EXAMPLE 5.7:

A graph G with a spanning tree in solid lines is shown in Fig. 5.11. Each of the dashed lines represents a nontree edge that generates a basis cycle relative to T. □

Suppose we are given a spanning tree T of our undirected and connected graph $G = (V, E)$. Define a relation R_c on the set E as follows: For any pair (e, g) of edges, eR_cg *if and only if* e *and* g *belong to a common cycle determined by a nontree edge*. The **reflexive transitive closure** R_c^* of R_c consists of all pairs of edges (e, g) for which either (1) $e = g$, or (2) there exist edges $f_1, f_2, \ldots, f_t$ such that $eR_cf_1, f_1R_cf_2, \ldots, f_{t-1}R_cf_t$, and f_tR_cg.

We are ready to establish the following lemma.

Lemma 5.5: *Let T be an arbitrary spanning tree of a connected graph $G = (V, E)$, and let R_c be the relation defined previously. Then, $R_b = R_c^*$, where R_b is the equivalence relation defining the blocks of G, and R_c^* is the reflexive, transitive closure of R_c.*

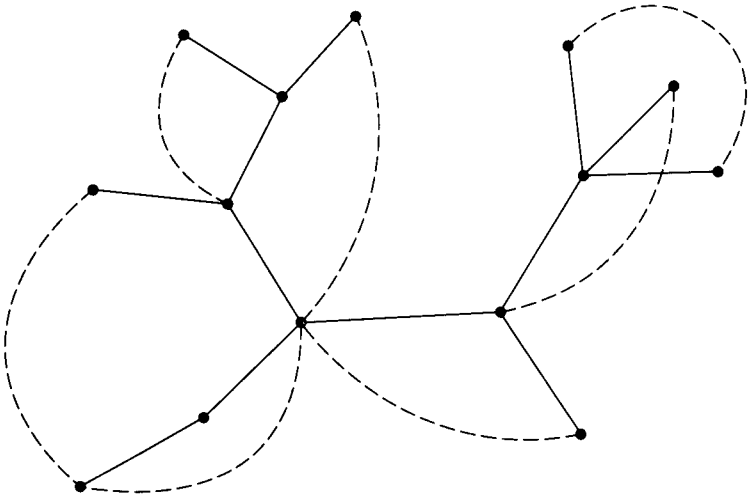


FIGURE 5.11
The spanning tree of the given graph is shown in solid lines. Each of the dashed lines creates a basis cycle. The collection of these cycles form a cycle basis for the given graph.

Proof: According to the definition of R_c, eR_cg implies that e and g belong to a common basis cycle relative to T, and hence that $R_c \subseteq R_b$. But since R_b is an equivalence relation, this result implies that $R_c^* \subseteq R_b$.

Let e and g be two edges such that eR_bg. If $e = g$, then we are done, since by definition R_c^* is reflexive. Otherwise, there exists a simple cycle C containing e and g. Let $\{C_i\}$ be the cycle basis determined by the spanning tree T. Then, $C = C_{i_1} \oplus \ldots \oplus C_{i_t}$, for some indices $i_1, i_2, \ldots, i_t$. Given any two cycles C_{i_j} and C_{i_k}, there exist cycles $C_{\alpha_1}, C_{\alpha_2}, \ldots, C_{\alpha_r}$ such that C_{i_j} shares an edge with C_{α_1}, C_{α_1} shares an edge with C_{α_2}, and so on, until we reach C_{α_r}, which shares an edge with C_{i_k}. If the cycles C_{α_j}'s do not exit, the exclusive OR $C_{i_1} \oplus \ldots \oplus C_{i_t}$ will not be a single cycle. Since the edges in each C_{i_j} are clearly in R_c, by transitivity, all the edges in C are in R_c^*, and therefore $eR_c^* g$. □

Suppose we can somehow identify R_c. Then, the blocks of G can be determined as follows. Let $G' = (V', E')$, where V' is the set E of edges of G, and $(e, g) \in E'$ if and only if eR_cg. The connected components of G' correspond to the equivalence classes of R_c^*, and hence uniquely identify the blocks of G.

EXAMPLE 5.8:

Consider again the graph G of Fig. 5.10; a spanning tree of G is shown in solid lines in Fig. 5.12(a). The corresponding cycle basis consists of the three cycles $C_1 = \{e_1, e_3, e_4\}$, $C_2 = \{e_2, e_4, e_5\}$, and $C_3 = \{e_8, e_9, e_{10}\}$. The graph G' of the relation R_c is shown in Fig. 5.12(b). For example, there is an edge connecting e_4 and e_2 because both edges belong to C_2, and there is no edge incident on e_6 or e_8 because neither edge is contained in any of the cycles C_1, C_2, and C_3. The connected components of G' are given by $\{e_1, e_2, e_3, e_4, e_5\}$, $\{e_8, e_9, e_{10}\}$, $\{e_6\}$, and $\{e_7\}$, which define precisely the blocks of G. □

Unfortunately, R_c may be too large for our purposes. Consider, for example, the case when G is a simple cycle. Then, $|R_c| = \Theta(n^2)$, in spite of the fact that $m = n$, in this case. We now define a subset of R_c, which will be shown to induce the same connected components as does R_c.

The relation R_c has been defined relative to a given spanning tree T of the input graph $G = (V, E)$. We wish to define a relation R_c' on the set E such that (1) $R_c' \subseteq R_c$, (2) $|R_c'| = O(m)$, and (3) $(R_c')^* = R_c^*$. To define R_c', we find it useful to number the vertices in V using a preorder traversal of the spanning tree T rooted at an arbitrary vertex r. A **preorder traversal** of T consists of a traversal of the root r, followed by the preorder traversals of the subtrees of r from left to right (also introduced in Exercise 3.13). For the remainder of this section, *we identify each vertex v by its preorder number.* We illustrate the usefulness of the preorder numbering with the following simple lemma.

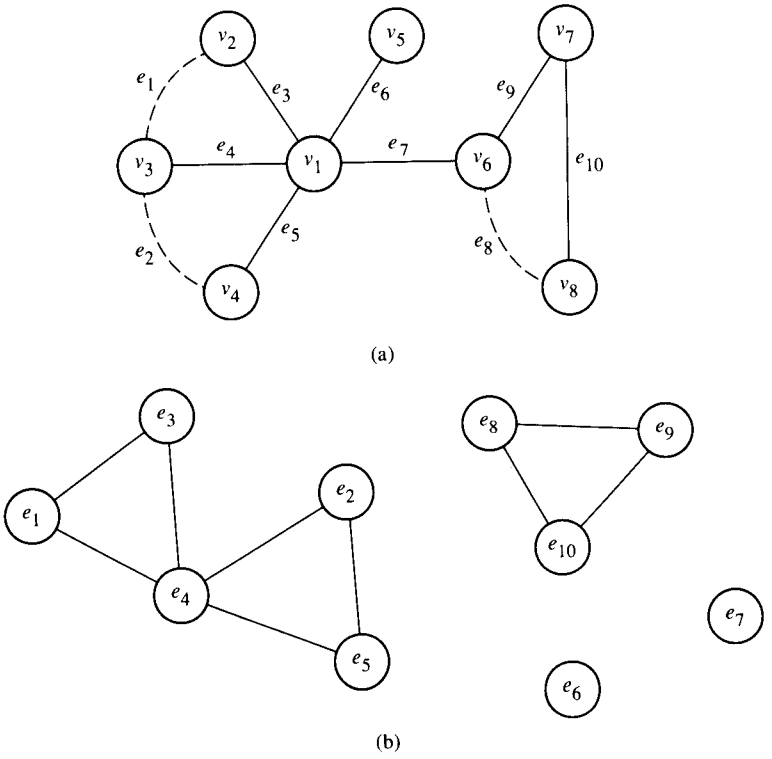


FIGURE 5.12
An illustration of the graph of the relation R_c defined by eR_cg if and only if e and g belong to a cycle determined by a nontree edge. (a) A spanning tree T shown in solid lines together with the remaining edges shown in dashed lines. (b) The graph of the relation R_c obtained from T; two edges e_i and e_j are connected if and only if they belong to a basis cycle relative to T. The connected components of the graph shown in (b) are the blocks of the graph shown in (a).

Lemma 5.6: *Let T be a rooted tree in which each vertex has been identified with its preorder number, and let size(v) denote the number of vertices in the subtree rooted at v. Then, the following claims hold:*

1. *Given any two vertices u and v, if $u < v$, then v cannot be an ancestor of u.*
2. *Given any two vertices u and v such that $v + size(v) \leq u$, then u and v are not related; that is, neither vertex is an ancestor of the other vertex.*

Proof: Claim1 follows from the fact that a vertex v is always visited before any of its descendants in a preorder traversal of the tree.

As for claim2, u cannot be an ancestor of v, since $v < u$. Any descendant w of v must be visited before the preorder traversal of the subtree rooted at v has been completed. Therefore, w must satisfy $v \leq w < v + size(v)$, which implies that vertex u cannot be a descendant of v. □

We are now ready to define the new relation R'_c. For each vertex u, we denote the parent of u in the rooted spanning tree T by $p(u)$. Given any two edges e and g of E, eR'_cg if and only if one of the following conditions hold:

1. $e = (u, p(u))$ and $g = (u, v)$ in $G - T$, and $v < u$ (Fig. 5.13a).
2. $e = (u, p(u))$ and $g = (v, p(v))$, and (u, v) in $G - T$ such that u and v are not related (Fig. 5.13b).
3. $e = (u, p(u) = v)$, $v \neq 1$, and $g = (v, p(v))$, and some nontree edge of G joins a descendant of u to a nondescendant of v (Fig. 5.13c).

EXAMPLE 5.9:

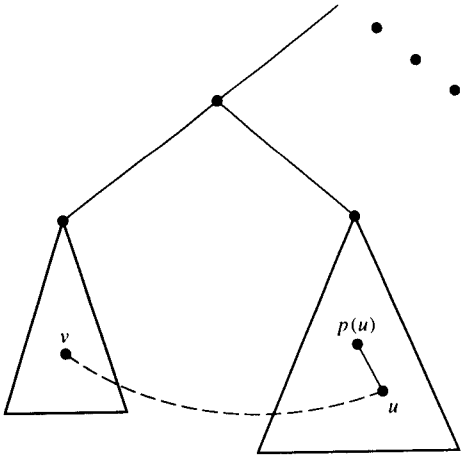
Consider again the graph G of Fig. 5.10(a). A possible rooted spanning tree is shown in Fig. 5.14(a). We now show how to determine the relation R'_c. By condition 1, we have that (e_4, e_1), (e_5, e_2), and (e_{10}, e_8) are in R'_c. By condition 2, we obtain that (e_5, e_4) and (e_4, e_3) are in R'_c. Condition 3 implies that (e_9, e_{10}) is in R'_c. The graph corresponding to R'_c is shown in Fig. 5.14(b). The connected components of G' correspond to the blocks of G. □

The following lemma provides a justification for our definition of R'_c.

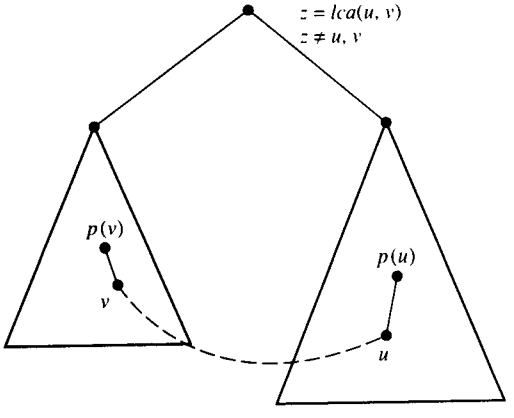
Lemma 5.7: *Let R_c and R'_c be the relations defined previously with respect to an arbitrary rooted spanning tree T of G. Then, $R_c^* = (R'_c)^*$, and $|R'_c| = O(m)$, where m is the number of edges in G.*

Proof: Any two edges e and g that satisfy eR'_cg belong to a cycle determined by a nontree edge. Hence, they are in R_c, which implies that $R'_c \subseteq R_c$ and thus $(R'_c)^* \subseteq R_c^*$.

We now prove the converse. Let e and g be two edges that satisfy $eR_c\,g$. Then, e and g lie on a common basis cycle relative to T—say, that defined by the nontree edge (u, v). Let z be the lowest common ancestor of u and v. We show that all the edges in the basis cycle are in $(R'_c)^*$, which will establish that $R_c^* \subseteq (R'_c)^*$.

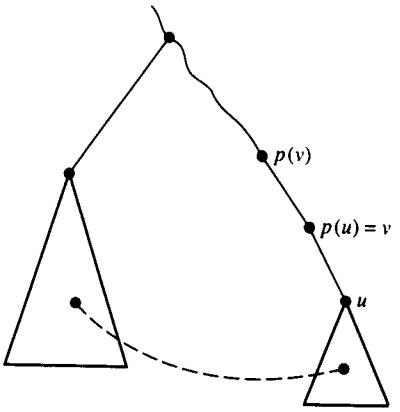


(a)



(b)

FIGURE 5.13
Pairs of edges defining the relation R'_c. (a) (u, v) is a nontree edge such that $v < u$; hence, the pair of edges (u, v) and $(u, p(u))$ belong to R'_c. (b) (u, v) is a nontree edge such that u and v are not related; hence, $(u, p(u))$ and $(v, p(v))$ belong to R'_c.



(c)

FIGURE 5.13 (*continued*)
(c) The dashed line represents a nontree edge connecting a descendant of u to a nondescendant of v; hence, the pair of edges $(u, p(u))$ and $(v, p(v))$ are in R'_c.

Assume without loss of generality that $u < v$. We consider two separate cases.

1. $z \neq u$ (see Fig. 5.15a): Hence, u and v are unrelated, and thus $(u, p(u))$ and $(v, p(v))$ are in R'_c by condition 2. Using condition 1, we obtain that $(v, p(v))$ and (v, u) are also in R'_c. By condition 3, we obtain that each pair of successive edges on the path between z and u, and on the path between z and v, is also in R'_c. By transitivity, we obtain that $e(R'_c)^* g$.
2. $z = u$ (see Fig. 5.15b): We apply condition 3 several times to obtain that e and g are in R'_c.

As for the size of R'_c, we bound the number of edges satisfying each of conditions 1, 2, and 3. Each edge in $G - T$ determines one pair of edges satisfying condition 1. As for condition 2, an edge in $G - T$ may determine at most one pair of edges in R'_c. Given the restriction on the pairs defined in condition 3, the total number of such pairs is no more than the number of edges in T. Therefore, $|R'_c| = O(m)$. □

Let $G' = (V', E')$ be the graph corresponding to R'_c.

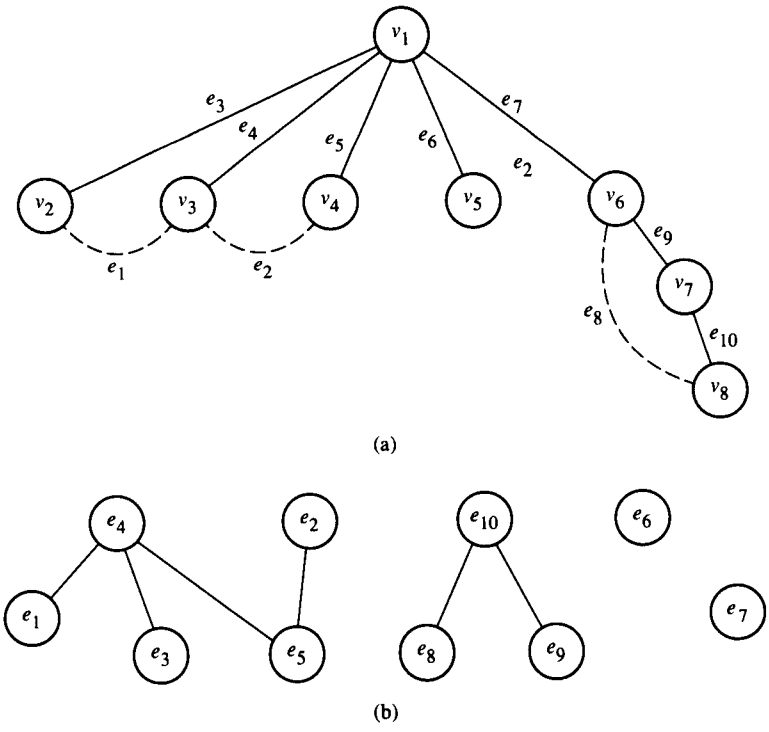


FIGURE 5.14
Determination of the relation R_c' for Example 5.9. (a) A spanning tree in solid lines; the remaining edges are in dashed lines. (b) The graph corresponding to the relation R_c'. The connected components of R_c' correspond exactly to the blocks of the original graph.

Corollary 5.1: *Let R_b and R_c' be the two relations defined previously. Then, $(R_c')^* = R_b$; hence, two edges of G are in a common block if and only if they are in a common connected component of the graph G' defined by R_c'.* □

5.3.2 THE BICONNECTED-COMPONENTS ALGORITHM

For each vertex v in the rooted spanning tree T of G, let $p(v)$ be the parent of v and let $size(v)$ be the number of vertices in the subtree rooted at v. We have already developed a parallel algorithm for computing this function in Section 3.2. We now introduce two new functions on the set V.

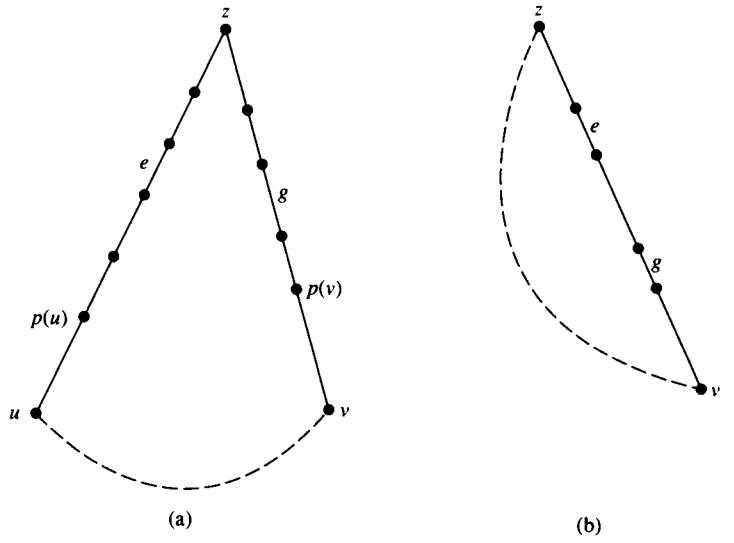


FIGURE 5.15
Illustration of the proof that the pair (e, g) of tree edges satisfies the relation $(R'_c)^*$. The vertex z is the lowest common ancestor of u and v, where (u, v) is the nontree edge that induces the basis cycle containing e and g. (a) The case when $z \neq u, v$. (b) The case when $z = u$.

For each $v \in V$, let $low(v)$ denote the smallest vertex that is either a descendant of v or adjacent to a descendant of v by a nontree edge. Similarly, $high(v)$ denotes the largest vertex that is either a descendant of v or adjacent to a descendant of v by a nontree edge. These two functions are useful in the determination of the pairs of edges satisfying condition 3 of the definition of R'_c, as shown by the following lemma.

Lemma 5.8: *Let* $e = (u, p(u) = v), v \neq 1$, *and* $g = (v, p(v))$ *be two consecutive tree edges of* T. *Then,* e *and* g *satisfy condition 3 of the definition of* R'_c *if and only if* $low(u) < v$ or $high(v) \geq v + size(v)$.

Proof: According to condition 3, e and g are in R'_c if some edge of E joins a descendant of u to a nondescendant of v.

Suppose that e and g satisfy condition 3, and let (x, y) be the edge joining the descendant x of u to the nondescendant y of v. There are two cases, depending on whether $y < u$ or $y > u$. We start with the case where $y < u$.

Since y is a nondescendant of v and $(u, p(u) = v)$ is an edge of T, we must have $y < v$. But $low(u) \leq y$, and hence $low(u) < v$. The second case is where $y > u$. Since y is a nondescendant of v and (u, v) is an edge of T, we must have that y is visited after the preorder traversal of the subtree rooted at v has been completed. Therefore, $y \geq v + size(v)$, which implies that $high(u) \geq v + size(v)$.

We now prove the converse. Let $e = (u, p(u) = v)$ and $g = (v, p(v))$ be a pair of tree edges satisfying $low(u) < v$ or $high(v) \geq v + size(v)$. Suppose that $low(u) < v$. Then, by the definition of $low(u)$, there exists a descendant x of u incident on a nontree edge (x, y) such that $y = low(u)$. Since $y = low(u) < v$, vertex y cannot be a descendant of v. Thus, (x, y) is an edge that connects a descendant of u to a nondescendant of v, and therefore e and g satisfy condition 3. The case where $high(u) \geq v + size(v)$ can be treated in a similar fashion. This concludes the proof of the lemma. □

We next develop a simple algorithm to compute the function $low(v)$ for each $v \in V$. The algorithm consists of two main steps. The first step assigns, to each vertex v, a weight, which is the minimum of v and any vertex connected to v by a nontree edge. The second step determines the minimum weight in the subtree rooted at v, for each vertex v. The details are stated in the next algorithm.

ALGORITHM 5.4
(Computing *low*(*v*))
Input: *A rooted spanning tree T of a connected graph G represented by a set of adjacency lists that support the Euler-tour technique.*
Output: *The value of low(v), for each vertex v.*
begin
1. For each vertex v, compute $w(v) = \min\{v\} \cup \{u \mid (v, u)$ *is a nontree edge*$\}$.
2. For each vertex v, $low(v) = \min\{w(u) \mid u$ *is in the subtree rooted at* $v\}$.
end

Lemma 5.9: *Algorithm 5.4 correctly computes low(v) for each vertex* $v \in V$. *The running time is* $O(\log n)$, *and the total number of operations is* $O(m)$, *where n is the number of vertices, and m is the number of edges.*

Proof: The correctness proof follows from the definition of the *low* function.

The time analysis is done under the assumption that each vertex of the input graph G contains a pointer to its adjacency list (the same as in the algorithms using the Euler-tour technique). Given the fact that the edges of

the spanning tree are marked, step 1 involves the computation of the minimum of unmarked elements in each list. Clearly, this step can be performed in $O(\log n)$ time, using a total of $O(m)$ operations. As we have seen in Section 3.3.3, the tree-contraction algorithm (Algorithm 3.7) can handle step 2 in $O(\log n)$ time, using $O(n)$ operations. □

The problem of computing, for all $v \in V$, the value $high(v)$ can be solved in a similar fashion using the same resource bounds.

We are ready to present the overall biconnected-components algorithm.

ALGORITHM 5.5
(Biconnected Components)

Input: *A connected, undirected graph G represented either by an adjacency matrix or by a sequence of edges.*
Output: *An array B such that* $B(e) = B(g)$ *if and only if e and g are in the same biconnected component. Each entry of B is the serial number of an edge of G.*
begin
1. *Determine a spanning tree T of G.*
2. *Root T at an arbitrary vertex, and identify each vertex with its preorder number.*
3. *For each vertex* $v \in V$, *compute* $size(v)$, $low(v)$, *and* $high(v)$.
4. *Construct a list L of pairs of edges as follows.*
   - *4.1 For each edge* $e = (u, v)$ *in* $G - T$ *such that* $v < u$, *put the pair* (e, g) *on L, where* $g = (u, p(u))$.
   - *4.2 For each edge* $(u, v) \in G - T$ *such that* $v + size(v) \leq u$, *put the pair* (e, g) *on L, where* $e = (v, p(v))$ *and* $g = (u, p(u))$.
   - *4.3 For each edge* $e = (u, p(u) = v)$, $v \neq 1$, *add* (e, g) *to L if* $low(u) < v$ *or* $high(u) \geq v + size(v)$, *where* $g = (v, p(v))$.
5. *Find the connected components of the graph determined by the edges on L. These components are identified by an array B.*

end

Theorem 5.4: *Algorithm 5.5 correctly identifies the edges in each biconnected component of the input graph. This algorithm can be implemented in* $O(\log n)$ *time, using* $O((m + n) \log n)$ *operations on the arbitrary CRCW PRAM.*

Proof: We start with the correctness proof. We only need to elaborate on step 4. We start by establishing the following claim.

Claim: The pairs of edges put on the list L after the execution of step 4 are precisely those pairs satisfying the relation R'_c.

Proof: Any pair generated in step 4.1 satisfies condition 1, and any pair that satisfies condition 1 is generated in step 4.1. Consider a pair of edges corresponding to an edge (u, v) in $G - T$ identified in step 4.2. As we have seen in Lemma 5.6, the condition $v + size(v) \leq u$ implies that u and v are not related. Therefore, condition 2 of the definition of R'_c applies to $(v, p(v))$ and $(u, p(u))$. Conversely, every pair of edges satisfying condition 2 of the definition of R'_c will be put on L in step 4.2. Using Lemma 5.8, we obtain that the pairs of edges identified in step 4.3 are exactly those pairs satisfying condition 3 of the definition of R'_c, and the proof of the claim is complete.

From Lemma 5.7, it follows that the edges in each connected component of the graph determined by L define a biconnected component of the original graph.

As for the complexity bounds, we estimate the resources of each step separately.

We can compute the spanning tree T of G (step 1) by using Algorithm 5.2, which runs in $O(\log n)$ time, using $O((m + n) \log n)$ operations on the arbitrary CRCW PRAM.

As for step 2, the Euler-tour technique can be used (1) to root T, (2) to compute the preorder number of each vertex, and (3) to compute $size(v)$, for each vertex v. As we have seen in Section 3.2, these functions can be computed in $O(\log n)$ time, using a linear number of operations, whenever T is represented by the adjacency lists with some additional pointers. Obtaining these adjacency lists from the marked edges of T can be done within the resources required for determining T (the details are left to Exercise 5.15).

The problem of computing the functions *high* and *low* can be done in $O(\log n)$ time, using $O(m)$ operations, as shown in Lemma 5.9.

Each of steps 4.1, 4.2, 4.3 can be executed in $O(1)$ time using $O(m)$ operations. Finally, we can handle step 5 by using the connected-components algorithm for sparse graphs (Algorithm 5.2), covered in Section 5.1. Therefore, the theorem follows. □

PRAM Model: Steps 2 and 3 of Algorithm 5.5 use the Euler-tour technique and hence do not require any simultaneous memory access. Steps 1 and 5 require the arbitrary CRCW PRAM. Step 4 needs only concurrent-read capability. □

Remark 5.8: We can modify Algorithm 5.5 slightly to obtain a CREW PRAM algorithm for determining the biconnected components in $O(\log^2 n)$ time, using $O(n^2)$ operations. The details are left to Exercise 5.17. □

5.4 Ear Decomposition

Graph-traversal methods are useful mainly because they induce a decomposition of the graph into a structured set of simple components. Depth-first search and breadth-first search are two such methods that have been found to be effective in handling many graph-theoretic problems. However, no efficient parallel implementations of these two methods are known at this point. To remedy the lack of such methods, we introduce the technique of **ear decomposition,** which does have an efficient parallel implementation.

An ear decomposition is essentially an ordered partition of the set of edges into simple paths (which include simple cycles). More formally, let $G = (V, E)$ be an undirected graph, with $|V| = n$ and $|E| = m$. Let P_0 be an arbitrary simple cycle of G. An ear decomposition of G starting with P_0 is an ordered partition of the set of edges $E = P_0 \cup P_1 \cup \ldots P_k$, such that, for each $1 \leq i \leq k$, P_i is a simple path whose endpoints belong to $P_0 \cup P_1 \cup \ldots \cup P_{i-1}$, but none of whose internal vertices does. Each simple path P_i is called an **ear.** If, for each $i > 0$, P_i is not a cycle, the decomposition is called an **open ear decomposition.**

EXAMPLE 5.10:

Consider the graph in Fig. 5.16. A possible ear decomposition is shown. All the edges belonging to ear P_i are labeled by the index i. Note that the example is not an open ear decomposition, since P_2 is a cycle. □

A characterization of the graphs with ear decompositions is given by the following theorem, the proof of which is left to Exercise 5.26.

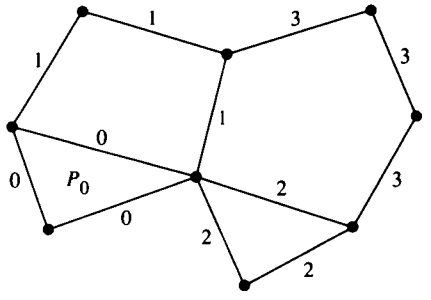


FIGURE 5.16
An ear decomposition of G consisting of four ears, such that the edges in ear P_i are all marked with the index i.

Theorem 5.5: *An undirected graph $G = (V, E)$ has an ear decomposition if and only if it is bridgeless (that is, there exists no edge whose removal disconnects the graph). The graph G has an open ear decomposition if and only if it is biconnected.* □

Let G have an *open* ear decomposition $E = P_0 \cup P_1 \cup \ldots \cup P_k$. Hence, P_0 is a simple cycle, P_1 is a simple path whose two distinct endpoints are on P_0, P_2 is a simple path whose two distinct endpoints are on $P_0 \cup P_1$, and so on. It is clear that such a decomposition yields an independent set of cycles. Actually, removing an arbitrary edge from each ear results in a spanning tree of G, as we can show by using an induction on the number of ears. Therefore, the number of ears is equal to the number $(m - n) + 1$ of nontree edges, and the ear decomposition yields a cycle basis for G. These facts hold for any (not necessarily open) ear decomposition.

On the other hand, we already know that a cycle basis can be generated from an arbitrary spanning tree of G by the nontree edges. This observation suggests the following method to obtain an ear decomposition.

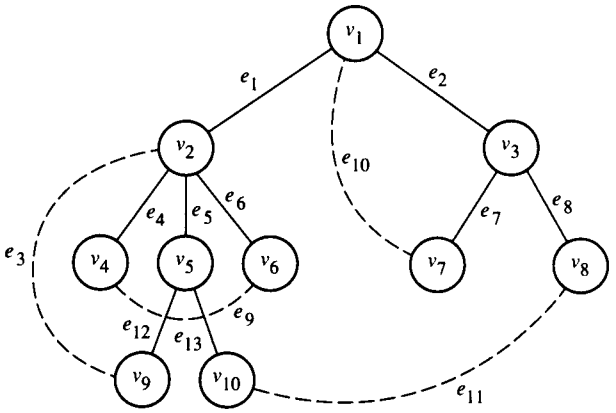
We start by computing a rooted spanning tree T of G. We would like to associate an ear P_e with each nontree edge e. Let C_e be the basis cycle induced by e. In general, we cannot take $P_e = C_e$, since edges in C_e may be shared by several other basis cycles. Hence, we have to break up C_e into paths, such that each path belongs to a single ear. One way to accomplish this goal is to label each nontree edge $e = (u, v)$ as follows. Let *level*(e) be the level of the lowest common ancestor of u and v; that is, $lca(e) = lca(u, v)$. Then, *label*(e) is defined to be the pair (*level*(e), $s(e)$), where $s(e)$ is the serial number of e and $1 \leq s(e) \leq m$. For each tree edge g, let *label*(g) be the smallest label of any nontree edge whose cycle contains g.

EXAMPLE 5.11:

Let G be the graph shown in Fig. 5.17. The dark lines represent the tree edges of a rooted spanning tree, and the dashed edges represent the nontree edges. Starting with the nontree edges, we obtain that *label*$(e_3) = (1, 3)$, *label*$(e_9) = (1, 9)$, *label*$(e_{10}) = (0, 10)$, *label*$(e_{11}) = (0, 11)$. Using this information, the labels of the tree edges are given by *label*$(e_1) = (0, 11)$, *label*$(e_2) = (0, 10)$, *label*$(e_4) = (1, 9)$, *label*$(e_5) = (0, 11)$, *label*$(e_6) = (1, 9)$, *label*$(e_7) = (0, 10)$, *label*$(e_8) = (0, 11)$, *label*$(e_{12}) = (1, 3)$, and *label*$(e_{13}) = (0, 11)$. □

We are ready to justify the introduced labels.

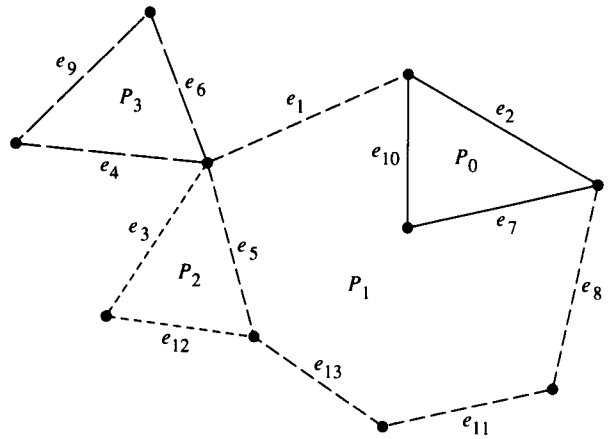
Lemma 5.10: *Let T be a rooted spanning tree of a graph $G = (V, E)$, and, for each edge $g \in E$, let label(g) be the function defined previously. Given any nontree edge e, let $P_e = \{e\} \cup \{g \in T \mid label(g) = label(e)\}$. Then P_e is a simple path (or cycle). Moreover, every tree edge belongs to exactly one such path.*



(a)

Edge	Label
e_1	(0, 11)
e_2	(0, 10)
e_3	(1, 3)
e_4	(1, 9)
e_5	(0, 11)
e_6	(1, 9)
e_7	(0, 10)
e_8	(0, 11)
e_9	(1, 9)
e_{10}	(0, 10)
e_{11}	(0, 11)
e_{12}	(1, 3)
e_{13}	(0, 11)

(b)



(c)

FIGURE 5.17
Ear decomposition for the graph of Example 5.11. (a) A graph G, where the tree edges are shown in solid lines and the remaining edges shown in dashed lines. (b) The labels of all the edges. (c) The decomposition determined by the ear-decomposition algorithm. The edges in each ear are drawn using a different type of lines. The cycle P_0 consists of all the edges with the label (0, 10), P_1 corresponds to the label (0, 11), P_2 corresponds to the label (1, 3), and P_3 corresponds to the label (1, 9).

Proof: Let the nontree edge be given by $e = (u, v)$. Consider the path Q in T from v to $lca(u, v)$, and let $v \neq lca(u, v)$. Let $h = (x, p(x))$ be the first edge on Q such that $label(h) \neq label(e)$ (see Fig. 5.18). The inequality of $label(e)$ and $label(h)$ implies the existence of a nontree edge (z, x) such that the label of (z, x) is smaller than $label(e)$. Hence none of the edges on Q between x and $lca(u, v)$ belongs to P. The argument is similar if we consider the path from u to $lca(u, v)$. Hence, P_e is a simple path.

Since the labels of the nontree edges are distinct, every tree edge belongs to exactly one such path. □

EXAMPLE 5.12:

The labels of all the edges in Fig. 5.17 were determined in Example 5.11. Grouping together all the edges with the same label, we obtain the following

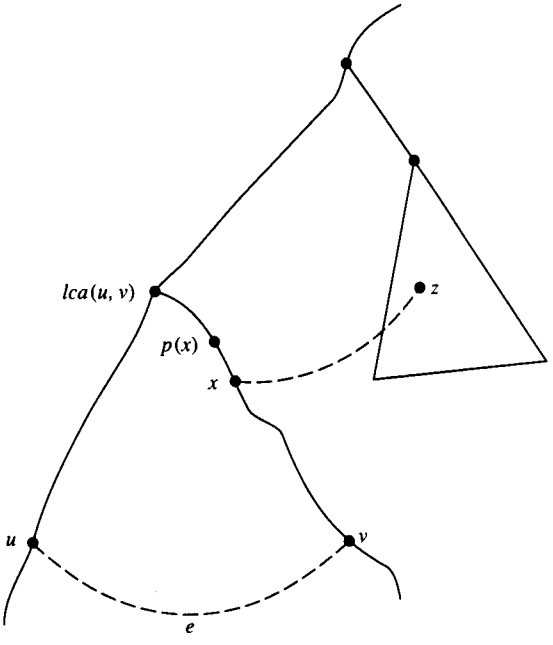


FIGURE 5.18
An illustration of the proof that all the edges with the same label as the nontree edge $e = (u, v)$ form a simple path. In this case, $h = (x, p(x))$ is the first edge on the path from v to $lca(u, v)$ such that $label(h) \neq label(e)$. All the edges from x to $lca(u, v)$ have a label that is smaller than $label(e)$.

sets: $P_0 = \{e_{10}, e_2, e_7\}, P_1 = \{e_1, e_5, e_8, e_{11}, e_{13}\}, P_2 = \{e_3, e_{12}\}$, and $P_3 = \{e_4, e_6, e_9\}$. Note that we have ordered these sets by their labels. It is easy to check that this sequence indeed defines an ear decomposition as shown in Fig. 5.17(c). This example is not an open ear decomposition, since P_3 is a simple cycle. □

In summary, for each nontree edge e, we defined a set P_e of edges, which turned out to form a simple path such that every tree edge belonged to exactly one such path. Therefore, the collection $\{P_e\}$ over the nontree edges e forms a decomposition of E into simple (possibly closed) paths. The fact that they can be ordered to yield an ear decomposition will be shown after the statement of the algorithm.

ALGORITHM 5.6
(Ear Decomposition)
Input: *A bridgeless, undirected graph G represented by a sequence of edges.*
Ouput: *An ordered set of paths representing an ear decomposition of G.*
begin
1. Find a spanning tree T of G.
2. Root T at an arbitrary vertex r, and compute level(v) and p(v), for each vertex v ≠ r, where level(v) and p(v) are the level and the parent of v, respectively.
3. For each nontree edge e = (u, v), compute lca(e) = lca(u, v) and level(e) = level(lca(e)). Set label(e): = (level(e), s(e)), where s(e) is the serial number of e.
4. For each tree edge g, compute label(g).
5. For each nontree edge e, set $P_e = \{e\} \cup \{g \in T \mid label(g) = label(e)\}$. Sort the P_e's by label(e).
end

Theorem 5.6: *Let $G = (V, E)$ be a bridgeless graph. Algorithm 5.6 correctly finds an ear decomposition of G. This algorithm can be implemented to run in $O(\log n)$ time, using a total number of $O((m + n) \log n)$ operations.*

Proof: We start with the correctness proof. As we saw in Lemma 5.10, each P_e is a simple path. We must now show that each of the path's endpoints belongs to some $P_{e'}$, where $label(e') < label(e)$, and that none of its internal vertices belongs to such an ear.

Claim: Given any nontree edge e such that its label is not the minimum label among the nontree edges, each of the endpoints of P_e belongs to some $P_{e'}$

satisfying $label(e') < label(e)$, and none of the internal vertices of P_e belongs to such an ear.

Proof of the Claim: Let e_0 be the nontree edge whose label is minimum; hence, $label(e_0) = (0, s(e_0))$, where $s(e_0)$ is the minimum over all edges e such that $lca(e)$ is the root. Such an edge must exist, because otherwise the graph G would have a bridge incident on the root. The ear P_{e_0} corresponding to e_0 is the basis cycle C_{e_0}.

Let e be an arbitrary nontree edge such that $label(e) > label(e_0)$. The proof of Lemma 5.10 indicates that either P_e is the basis cycle C_e or each of the endpoints belongs to some edge g such that $label(g) < label(e)$. In the latter case, g has a smaller label, and hence the proof follows. In the case when $P_e = C_e$, let v be $lca(e)$. If v is the root, then we are done, since C_e intersects the ear C_{e_0} at the root. Otherwise, consider the edge $(v, p(v))$, which must belong to some C_f, for some nontree edge f; if it does not, then $(v, p(v))$ will be a bridge, which is impossible. The level of $lca(f)$ must be smaller than $level(v)$; hence, v belongs to an ear whose label is smaller than $label(e)$. Therefore, the claim follows.

We now estimate the complexity bounds of Algorithm 5.6.

Step 1 requires a spanning-tree algorithm, which can be derived from the connected-components algorithm for sparse graphs (Algorithm 5.2). The corresponding running time is $O(\log n)$, and the total number of operations is $O((n + m) \log n)$. Step 2 can be done by the Euler-tour technique after the input representation has been modified to support this technique (Section 3.2.1). Step 3 requires the LCA algorithm developed in Section 3.4, and can be performed within the asymptotic bounds of step 1.

Step 4 can be done as follows. For each vertex v, let $f(v) = \min\{label(v, u) \mid (v, u) \text{ in } G - T\}$. Let $g = (x, y) \in T$, where $y = p(x)$. Then, $label(g)$ is the minimum w value in the subtree rooted at x. These two substeps can be executed in $O(\log n)$ time, using $O(m)$ operations, in a fashion similar to that of computing $low(v)$ (Lemma 5.9). Step 5 involves sorting the edges by their labels. The pipelined merge-sort algorithm (Algorithm 4.4) can be used to sort these edges in $O(\log n)$ time, using $O((m + n) \log n)$ operations. Therefore, the theorem follows. □

PRAM Model: We can find a spanning tree with $O(m \log n)$ operations by using Algorithm 5.2, which requires the arbitrary CRCW PRAM. Steps 2, 3, 4, and 5 of Algorithm 5.6 do not require any simultaneous memory access. If T is given as a part of the input, and the ears need not be sorted, the algorithm runs in $O(\log n)$ time, using a total of $O(m + n)$ operations and requiring no simultaneous memory access. □

Applications of the ear-decomposition algorithm (Algorithm 5.6) are given in Exercises 5.25 and 5.28.

5.5 Directed Graphs

We have seen a number of techniques that can be used to solve many problems on undirected graphs efficiently. In contrast, there are almost no such techniques for directed graphs. In this section, we introduce efficient techniques to handle two basic problems on directed graphs: *transitive closure* and *all pairs shortest paths*. These techniques can be used to handle many other basic problems on directed graphs. Unfortunately, the corresponding algorithms use an excessive number of operations compared to the best known sequential algorithms, and it seems that novel techniques are needed to handle basic directed-graph problems more efficiently.

Our two basic problems can be defined as follows. The **transitive closure** of a directed graph $G = (V, A)$ is the graph $G^* = (V, A^*)$, where A^* consists of all pairs $\langle i, j \rangle$ such that either $i = j$ or there exists a directed path from i to j.

Given a *weighted* directed graph $G = (V, A)$ with the weight function $w : A \rightarrow \mathcal{R}$, where $\mathcal{R}$ is the field of real numbers, the **all pairs shortest paths** problem is to compute, for every pair i and j of vertices, a "shortest" path from i to j; that is, we compute a directed path from i to j, if it exists, such that the total weight of its arcs is minimum over all such paths.

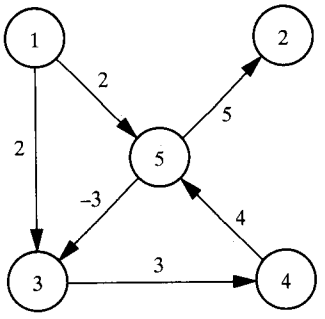
We represent a directed graph $G = (V, A)$ with n vertices by an $n \times n$ **incidence matrix** B such that (1) $B(r, s) = 1$ if $\langle r, s \rangle \in A$, and (2) $B(r, s) = 0$ otherwise. If G is a weighted graph, then we can represent it by its $n \times n$ **weight matrix** W such that

$$W(r, s) = \begin{cases} 0 & r = s; \\ w(\langle r, s \rangle) & r \neq s \text{ and } \langle r, s \rangle \in A; \\ \infty & r \neq s \text{ and } \langle r, s \rangle \notin A. \end{cases}$$

EXAMPLE 5.13:

A weighted directed graph $G = (V, A)$ together with its incidence matrix B and its weight matrix W are shown in Fig. 5.19(a). Fig. 5.19(b) shows the transitive closure of G, and Fig. 5.19(c) shows the distance matrix D such that $D(i, j)$ is the weight of a shortest path from i to j. Consider, for example, the case where $i = 1$ and $j = 3$. There are two directed paths from vertex 1 to vertex 3; the first consists of the arc $\langle 1, 3 \rangle$ and has a weight equal to 2, and the second consists of the two arcs $\langle 1, 5 \rangle$ and $\langle 5, 3 \rangle$ and has a weight equal to -1. Hence, the latter path is the shortest path yielding $D(1, 3) = -1$. □

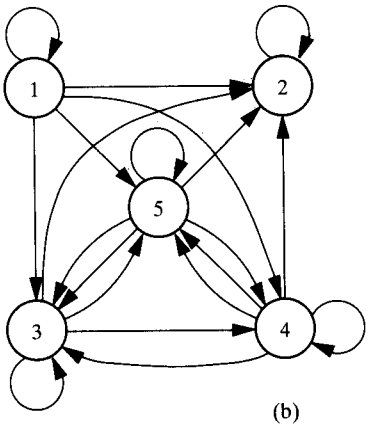
Since our directed graphs will be represented by matrices, the computation of a matrix product will be an important tool to solve our two main problems. We start with a discussion of the matrix-multiplication problem.



$$B = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

$$W = \begin{bmatrix} 0 & \infty & 2 & \infty & 2 \\ \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & 0 & 3 & \infty \\ \infty & \infty & \infty & 0 & 4 \\ \infty & 5 & -3 & \infty & 0 \end{bmatrix}$$

(a)



(b)

$$D = \begin{bmatrix} 0 & 7 & -1 & 2 & 2 \\ \infty & 0 & \infty & \infty & \infty \\ \infty & 12 & 0 & 3 & 7 \\ \infty & 9 & 1 & 0 & 4 \\ \infty & 5 & -3 & 0 & 0 \end{bmatrix}$$

(c)

FIGURE 5.19
The graph for Example 5.13. (a) A weighted directed graph, its incidence matrix B, and its weight matrix W. (b) The transitive closure of G. (c) The distance matrix of the weights of the shortest paths between all pairs of vertices.

5.5.1 MATRIX MULTIPLICATION

Let $A = (a_{ij})$ and $B = (b_{ij})$ be two $m \times n$ and $n \times p$ matrices, respectively. Let $C = AB$ be the product of A and B; that is, the (i,j) entry of C is defined by $c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}$, where $1 \le i \le m$ and $1 \le j \le p$. The obvious parallel implementation consists of computing all the products $a_{ik}b_{kj}$ for all i, k, j, and then using a balanced binary tree to compute each c_{ij}. This approach yields an $O(\log n)$ time algorithm, using a total of $O(mnp)$ operations running on the

CREW PRAM model. This algorithm was covered in detail in Section 1.3.2. In general, we cannot decrease the total number of operations or make the parallel algorithm faster without additional information about the entries of the matrices. In what follows, we consider two special cases for which such an improvement is possible.

Matrix Multiplication Over a Ring. A **ring** consists of a set S of elements together with two binary operations $+$ and $\cdot$ such that (1) $+$ and $\cdot$ are commutative and associative, (2) $\cdot$ is distributive over $+$, (3) each of $+$ and $\cdot$ has an identity element, and (4) each element $e \in S$ has an additive inverse. A typical ring is the set of integers together with the usual addition and multiplication operations. We can exploit the algebraic structure of a ring to reduce the number of arithmetic operations required to multiply two matrices.

Suppose that the elements of the matrices A and B belong to a ring R. Then it is possible to multiply A and B by using considerably fewer arithmetic operations than the $\Theta(mnp)$ operations required by the standard algorithm. The discussion of such algorithms is beyond the scope of this book. We mention here only that the best known bound for the case when $m = n = p$ is $O(n^{2.376})$ (see bibliographic notes for a reference). All the known fast sequential algorithms have a structure that is similar to that of the standard algorithm. More precisely, a general entry c_{ij} of the product $C = AB$ is generated as a linear combination of *bilinear multiplications*, where each such multiplication is of the form $L(a_{ij}) \times L'(b_{kl})$ and L, L' are linear forms over the ring R. The important point for us is that the fast matrix-multiplication algorithms lead to $O(\log n)$ time parallel algorithms using the same total number of operations.

We use $M(n)$ to denote the number of operations used by the best known sequential algorithm to compute the product of two $n \times n$ matrices. Then, from our discussion, we can compute the product of two $n \times n$ matrices in $O(\log n)$ time, using $O(M(n))$ operations on a CREW PRAM.

Boolean Matrix Multiplication. We now consider the case when the matrices are Boolean, and hence matrix multiplication is defined over the Boolean OR and AND operations. That is, the (i,j) entry of the product of A and B is the Boolean OR of the n AND terms $a_{ik}b_{kj}$, where $1 \leq k \leq n$. Each such Boolean OR can be computed in $O(1)$ steps, using a total of $O(n)$ operations on a common CRCW PRAM. Hence, Boolean matrix multiplication can be done in $O(1)$ time, using a total of $O(mnp)$ operations.

It can also be shown that Boolean matrix multiplication can be embedded in the ring of integers modulo $(n + 1)$; hence, the matrix multiplication bounds over a ring apply to Boolean matrix multiplication as well.

Matrix Powers. Another important matrix computation related to matrix multiplication is to compute A^{2^s}, where A is an $n \times n$ matrix, and s is a positive integer. The method of **repeated squaring** consists of iterating s times the computation $B = B^2$, where $B = A$ initially. Hence, we obtain successively $B = A, B = A^2, B = A^4, \cdots$, and finally $B = A^{2^s}$.

The resource requirements of the repeated-squaring method depend on the matrix-multiplication algorithm used to perform the squaring. In particular, the repeated-squaring method leads to an $O(s \log n)$ time algorithm, using $O(sM(n))$ operations, whenever the entries of A are over a ring. In the case where A is Boolean, we can compute A^{2^s} in $O(s)$ time, using $O(sn^3)$ operations.

We summarize our discussions concerning matrix products and powers in the following theorem.

Theorem 5.7: *Let A and B be two $n \times n$ matrices. Then, the following statements hold.*

1. *If the entries of A and B belong to a ring R, then the product AB can be computed in $O(\log n)$ time, using a total of $O(M(n))$ operations on the CREW PRAM, where $M(n)$ is the best known sequential bound for computing AB over R.*
2. *If A and B are Boolean matrices, then the product AB can be computed within the same bounds as in case 1, or in $O(1)$ time, using $O(n^3)$ operations on the common CRCW PRAM.*
3. *The problem of computing A^{2^s} can be solved in $O(s \log n)$ time, using a total of $O(sM(n))$ operations, if the entries belong to a ring. If A is Boolean, we can also compute A^{2^s} in $O(s)$ time, using a total of $O(sn^3)$ operations on the common CRCW PRAM.* □

5.5.2 TRANSITIVE CLOSURE

Let $G = (V,A)$ be a directed graph represented by its incidence matrix B. Let B^* be the incidence matrix of the transitive closure of the graph G. Since the entries of these incidence matrices are either 0 or 1, we can view them as Boolean matrices. The following theorem reduces the computation of B^* to computing a power of a Boolean matrix.

Theorem 5.8: *Let B be an $n \times n$ Boolean matrix representing the directed graph $G = (V,A)$. Then, the incidence matrix B^* of the transitive closure of the graph G is given by $B^* = (I + B)^{2^{\lceil \log n \rceil}}$, where I is the $n \times n$ identity matrix.*

Proof: Denote the matrix $(I + B)^s$ by $T^{(s)} = (t_{ij}{}^{(s)})$. In particular, $T^{(1)} = (t_{ij}{}^{(1)}) = I + B$. We establish the following induction hypothesis, which will essentially prove the theorem.

Induction Hypothesis: *For each positive integer s,* $t_{ij}^{(s)} = 1$ *if and only if there exists a path from i to j of length less than or equal to s, for all pairs* $1 \leq i, j \leq n$.

Proof: The base case corresponding to $s = 1$ follows trivially, since $T^{(1)} = I + B$ and hence $t_{ij}^{(1)} = 1$ if and only if either $i = j$ or $\langle i, j \rangle \in A$.

Assume that the induction hypothesis holds for all values less than or equal to s, where $s > 1$. We next show that the same holds true for $s + 1$.

Assume that an arbitrary entry $t_{ij}^{(s+1)}$ of $(I + B)^{s+1}$ is equal to 1. Since $(I + B)^{s+1} = (I + B)^s(I + B) = (I + B)^s T^{(1)}$, we have that $t_{ij}^{(s+1)} = \sum_{l=1}^{n} t_{il}^{(s)} t_{lj}^{(1)} = 1$, which implies that $t_{ik}^{(s)} = t_{kj}^{(1)} = 1$ for some k such that $1 \leq k \leq n$. The equality $t_{kj}^{(1)} = 1$ implies that either $k = j$ or $\langle k, j \rangle \in A$. By the induction hypothesis, the fact that $t_{ik}^{(s)} = 1$ implies that there exists a directed path from i to k of length at most s. Combining these two observations, we obtain that a path from i to j of length at most $s + 1$ must exist.

Conversely, assume that a certain path P of length at most $s + 1$ exists from i to j. We show that $t_{ij}^{(s+1)}$ must be equal to 1. The claim follows trivially if $i = j$, since $t_{ii}^{(1)} = 1$. Assume that $i \neq j$, and let k be the last vertex on P just before vertex j. Hence, P consists of a path P' from i to k of length at most s and the arc $\langle k, j \rangle \in A$. By the induction hypothesis, the existence of P' implies that $t_{ik}^{(s)} = 1$; moreover, the fact that $\langle k, j \rangle \in A$ implies that $t_{kj}^{(1)} = 1$. Therefore, $t_{ij}^{(s+1)} = \sum_{l=1}^{n} t_{il}{}^{(s)} t_{lj}{}^{(1)} = 1$, and the induction hypothesis follows.

Notice that a path from a vertex i to a vertex j exists if and only if there exists such a path of length less than or equal to $n - 1$. Combining this observation with the induction hypothesis implies that $B^* = (I + B)^m$, for any $m \geq n - 1$; hence, the theorem follows. □

Corollary 5.2: *The transitive closure of a directed graph with n vertices can be computed in* $O(\log n)$ *time, using* $O(n^3 \log n)$ *operations on the common CRCW PRAM, or in* $O(\log^2 n)$ *time, using* $O(M(n) \log n)$ *operations on the CREW PRAM, where* $M(n)$ *is the best known sequential bound for multiplying two* $n \times n$ *matrices over a ring.* □

5.5.3 ALL PAIRS SHORTEST PATHS

Let $G = (V, A)$ be a weighted directed graph represented by its weight matrix W. The graph G is assumed to have no cycles of negative weights. Our goal in this section is to develop a fast parallel algorithm to compute the $n \times n$

distance matrix D such that $D(i, j)$ is equal to the weight of a shortest path from i to j if that path exists; otherwise, $D(i,j)$ will be set equal to the value ∞.

We can generalize the transitive-closure algorithm developed in Section 5.5.2 to compute the matrix D by properly interpreting matrix multiplication. To give the background for our algorithm, we start by describing a graph-theoretic interpretation.

Let $D^{(s)}$ be an $n \times n$ matrix such that $D^{(s)}(i,j)$ is the minimum weight of a directed path from i to j that contains at most s arcs, for $s \geq 1$. For convenience, we introduce the matrix $D^{(0)}(i,j)$ as follows:

$$D^{(0)}(i,j) = \begin{cases} 0 & i = j; \\ \infty & i \neq j. \end{cases}$$

Next, we derive an expression for $D^{(s+1)}$ in terms of $D^{(s)}$ and the weight matrix W. A minimum-weight path from i to j containing at most $s + 1$ arcs is either of (1) length exactly $s + 1$ or (2) length less than or equal to s. In case 1, such a path consists of a subpath from i to some vertex k of length s and the arc $\langle k, j\rangle$; hence, its weight is given by $D^{(s)}(i, k) + W(k, j)$ (see Fig. 5.20). In case 2, the weight of the path is simply $D^{(s)}(i,j)$. Therefore, we conclude that

$$D^{(s+1)}(i,j) = \min\left(D^{(s)}(i,j), \min_{1 \leq k \leq n}\{D^{(s)}(i,k) + W(k,j)\}\right),$$

which is equivalent to

$$D^{(s+1)}(i,j) = \min_{1 \leq k \leq n}\{D^{(s)}(i,k) + W(k,j)\}, \tag{5.1}$$

since $W(j,j) = 0$ for all values of j.

Equation 5.1 is similar to the expression we encountered in defining a general entry of the product of two matrices (Section 5.5.1). In fact, if we replace the minimum and the sum operators by the sum and the multiplication operators, respectively, Eq. 5.1 becomes identical to the matrix-multiplication expression for multiplying $D^{(s)}$ and W. Therefore, we have the following lemma.

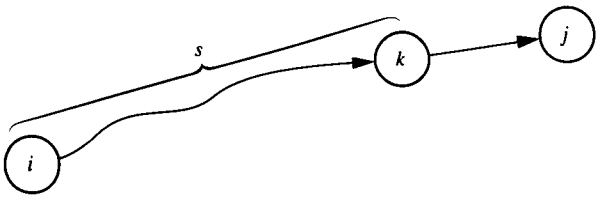


FIGURE 5.20
A minimum-weight path from i to j containing $s + 1$ arcs consists of a minimum-weight path from i to k containing s arcs and the arc $\langle k, j\rangle$.

Lemma 5.11: *Let $G = (V, A)$ be a weighted directed graph given by its weight matrix W. Given a positive integer s, let $D^{(s)}$ be the $n \times n$ matrix such that, for all pairs i and j, $D^{(s)}(i, j)$ is equal to the minimum weight of a path from i to j that at most contains s arcs. Then, $D^{(s+1)}(i,j) = \min_{1 \le k \le n} \{D^{(s)}(i,k) + W(k,j)\}$ for all $1 \le i, j \le n$, which can also be rewritten as $D^{(s+1)} = D^{(s)} \cdot W$, where matrix multiplication is carried over the two operations $\{min, +\}$.* □

The multiplication of two $n \times n$ matrices, where the operations min and $+$ have been substituted for $+$ and $\times$, respectively, can be carried out as in the standard matrix-multiplication algorithm. This technique results in $O(\log n)$ time parallel algorithm, using $O(n^3)$ operations.

Using Lemma 5.11, we have $D^{(1)} = D^{(0)} \cdot W$, which can be verified to be equal to W. Hence, $D^{(s)} = W^s$, for all values of s. Since we are assuming that our graph does not have a negative weight cycle, there exist simple shortest paths (if a shortest path crosses itself, the loop can be removed without its weight being increased). Therefore, we need only to compute $D^{(n-1)}$; hence, we have the following theorem.

Theorem 5.9: *Let W be the $n \times n$ weight matrix of a directed graph $G = (V, A)$. The matrix D of the weights of all pairs of shortest paths is given by $D = (W)^{2^{\lceil \log n \rceil}}$, where matrix multiplication is defined over the operators* {min, +}. □

Corollary 5.3: *Given the $n \times n$ weight matrix of a directed graph, the matrix D of the weights of all pairs of shortest paths can be computed in $O(\log^2 n)$ time, using a total of $O(n^3 \log n)$ operations.* □

5.6 Summary

We have developed in this chapter parallel algorithms for solving a few sample problems from graph theory. For undirected graphs, we have exploited the parallel techniques developed in earlier chapters for processing trees, both directed and undirected, in combination with problem-specific methods to obtain our efficient parallel algorithms. The parallel algorithms described for transitive closure and shortest paths follow directly from well-known sequential (and more general) algorithms.

Table 5.1 provides a summary of the asymptotic bounds of the main algorithms covered in this chapter.

TABLE 5.1
ALGORITHMS PRESENTED IN THIS CHAPTER.

Algorithm		Section	Time	Work*	PRAM Model
5.1	Connected Components for Dense Graphs	5.1.2	$O(\log^2 n)$	$O(n^2)$	Common CRCW
5.2	Connected Components for Sparse Graphs	5.1.3	$O(\log n)$	$O((m + n) \log n)$	Arbitrary CRCW
5.3	Minimum Spanning Tree	5.2.2	$O(\log^2 n)$	$O(n^2)$	CREW
5.4	Computing the Low Function	5.3.2	$O(\log n)$	$O(m)$	EREW
5.5	Biconnected Components	5.3.2	$O(\log n)$	$O((m + n) \log n)$	Arbitrary CRCW
5.6	Ear Decomposition	5.4	$O(\log n)$	$O((m + n) \log n)$	Arbitrary CRCW
	Transitive Closure	5.5.2	$O(\log n)$	$O(n^3 \log n)$	Common CRCW
	Transitive Closure	5.5.2	$O(\log^2 n)$	$O(M(n) \log n)$	CREW
	All Pairs Shortest Paths	5.5.3	$O(\log^2 n)$	$O(n^3 \log n)$	CREW

*n, number of vertices; m, number of edges; $M(n)$, best known sequential bound for $n \times n$ matrix multiplication

Exercises

5.1. The implementation of the connected components algorithm for dense graphs (Algorithm 5.1) given in the text requires the common CRCW PRAM model for constructing the adjacency matrix of the supervertex graph at each iteration. Show how to implement this algorithm on a CREW PRAM model without any asymptotic loss in efficiency.

5.2. Suppose you are given an undirected graph $G = (V, E)$ together with a vector D such that $D(u) = D(v)$ if and only if u and v are in the same connected component. Develop an efficient parallel algorithm to determine the number k of connected components, and to generate a separate list of the vertices in each connected component.

5.3. Fill in the details in the **while** loop of Algorithm 5.1 necessary to execute step 3 properly. Write the details to implement step 3 of this algorithm.

5.4. Let $D : \{1, 2, \ldots, n\} \rightarrow \{1, 2, \ldots, n\}$ be a function that induces a pseudoforest $\mathcal{F}$. Develop an $O(\log n)$ time algorithm to compute, for each $1 \leq i \leq n$, the minimum element in the directed tree containing i.

Your algorithm should run on the CREW PRAM model. What is the total number of operations used?

5.5. Suppose that step 2.2 of Algorithm 5.1 is modified so that $C(v)$ is set equal to any adjacent vertex, if there is one. Show that the modified algorithm will work correctly. Establish the parallel time and the number of operations required by this algorithm.

5.6. A given undirected graph $G = (V, E)$ is represented by its adjacency lists such that $|V| = n$ and $|E| = m$. Apply the same strategy as that used in the connected-components algorithm for dense graphs (Algorithm 5.1) to determine the connected components of G. The running time of your algorithm must be $O(\log^2 n)$. What is the total number of operations used, as a function of n and m?

5.7. Suppose we initialize the D function of the connected-components algorithm for sparse graphs (Algorithm 5.2) to $D(v) = v$, for all vertices v, without creating dummy vertices. Does the resulting algorithm work correctly? Explain your answer.

5.8. Suppose that we omit step 2 of the connected-components algorithm for sparse graphs (Algorithm 5.2). Provide an example for which this algorithm will require $\Omega(n)$ iterations to terminate. What is the number of iterations required to handle your example if step 2 is inserted back into the algorithm? Suppose that we omit step 1 of Algorithm 5.2; does Theorem 5.2 hold in this case? Justify your answer.

5.9. Let T be a directed tree corresponding to a function D on n elements. Develop an $O(1)$ time algorithm to check whether T is a rooted star. Your algorithm must use a linear number of operations. Specify the PRAM model used.

5.10. We have seen three different ways to represent a graph $G = (V, E)$: the adjacency matrix, a set of adjacency lists, and an unordered set of edges. Develop an $O(\log n)$ time parallel algorithm to convert one representation into another, where $|V| = n$. What is the total number of operations used in each case.

5.11. *An improved method for finding the connected components of a sparse graph is the following. During an iteration, only a subset of the edges is selected for the purpose of combining supervertices. Let n_i be the number of vertices in the nontrivial supervertex containing the fewest number of vertices during the ith iteration. Let $d = \sqrt{n_i}$. For each supervertex, select d edges connecting the supervertex to *different* supervertices. If the number of such edges is less than d, select the maximum possible such edges. Run Algorithm 5.2 $O(\log d)$ times, and then shrink each rooted directed tree into a rooted star. Repeat until

the number of vertices is less than $n/\log n$ and the number of edges is less than $O(m/\log n)$.

Show how to implement this strategy to obtain a parallel algorithm that runs in $O(\log n)$ time, using a total of $O((m + n) \log \log n)$ operations, where n is the number of vertices and m is the number of edges.

5.12. Prove Lemma 5.4.

5.13. Develop an algorithm to determine an MST of a connected weighted graph with n vertices and m edges in $O(\log n)$ time, using $O(m \log n)$ operations. *Hint:* Use the priority CRCW PRAM.

5.14. Given an undirected, connected graph G with n vertices, develop a detailed $O(\log n)$ time algorithm to determine an arbitrary spanning tree of G. The total number of operations used by your algorithm must be $O((n + m) \log n)$, where m is the number of edges of G. Assume that the input is given as a sequence of edges. Do not use the priority CRCW PRAM for your algorithm.

5.15. Let $T = (V, E_T)$ be a spanning tree of an undirected connected graph $G = (V, E)$. Suppose that T is represented by a sequence of edges given in an arbitrary order. Develop a parallel algorithm to determine the adjacency lists necessary to apply the Euler tour technique on T. What are the resources required by your algorithm?

5.16. An undirected graph $G = (V, E)$ is *bipartite* if there exists partition of V—say, $V = V_1 \cup V_2$—such that each edge has one endpoint in V_1 and the other in V_2. Develop an $O(\log n)$ time algorithm to find such a partition whenever it exists. *Hint:* Start by determining a spanning tree of G.

5.17. Modify the biconnected-components algorithm (Algorithm 5.5) such that it can be implemented on the CREW PRAM in $O(\log^2 n)$ time, using $O(n^2)$ operations. *Hint:* Modify steps 4 and 5 such that the edges determined in step 4.1 are not initially included in step 4.

5.18. A **cut vertex** v of a connected, undirected graph $G = (V, E)$ is a vertex whose removal leaves G disconnected.

a. Show that v is a cut vertex if and only if it belongs to more than one block.

b. Develop an efficient parallel algorithm to determine all the cut vertices of G.

5.19. Let $G = (V, E)$ be a connected graph. Recall that a **bridge** is an edge e whose removal leaves G disconnected. Develop a parallel algorithm to determine all the bridges of G. The complexity bounds of your algorithm should match those of either of the connected-components algorithms (Algorithms 5.1 and 5.2). *Hint:* Start by determining a spanning tree of G.

5.20. Let $G = (V, E)$ be an undirected graph given by its adjacency lists. An **Euler partition** is a partition of E into edge-disjoint paths (including cycles) $\{P_i\}$ such that each vertex of odd degree is the endpoint of exactly one path (with two distinct endpoints), and no vertex of even degree can appear as the endpoint of such a path. Show how to obtain an Euler partition in $O(\log n)$ time, using a linear number of operations. *Hint:* Start by making the graph Eulerian.

5.21. Let $G = (V, E)$ be a bipartite graph whose maximum degree $\Delta = 2^l$, for some positive integer l.

a. Show how to divide G into two bipartite graphs G_1 and G_2, each of maximum degree $\frac{\Delta}{2}$. Your algorithm should run in $O(\log n)$ time, using $O(|E| + |V|)$ operations. *Hint:* Determine an Euler partition, as defined in Exercise 5.20.

b. A k-edge coloring of G is an assignment $c : E \to \{1, 2, \ldots, k\}$ such that $c(e) \neq c(g)$ whenever e and g share a vertex. Use the algorithm for part (a) to find a Δ-edge coloring of G. State the resource bounds needed by your algorithm.

5.22. Let $G = (V, E)$ be a directed graph such that the indegree of each vertex is equal to the outdegree. Assume that, for each vertex v, the outgoing and the incoming edges are given by two ordered lists.

a. Show how to obtain a partition of E into a set of edge-disjoint cycles $\{C_i\}$ in $O(\log n)$ time, using a linear number of operations.

b. Let $G' = (V', E')$ be the undirected graph defined as follows. V' corresponds to the set of cycles obtained in (a) and $(C_i, C_j) \in E'$ if there exist $e \in C_i$ and $f \in C_j$ such that e and f are incoming edges to the same vertex—say, v—and f comes after e in the list of incoming arcs of v. Show that G' is connected.

c. A *swap* of two edges $e = \langle u, w \rangle$ and $f = \langle v, w \rangle$ in cycles C_i and C_j is an interchange of the edges' successors on these cycles. Prove that, for any given spanning tree T' of G', if we execute the swaps corresponding to the edges of T' in any order, all the cycles will be merged into an Euler circuit.

d. Develop an efficient algorithm to find an Euler circuit in G.

5.23. A **trail** of a connected graph G is a sequence $W = (e_1 = (v_0, v_1), e_2 = (v_1, v_2), \ldots, e_{k-1} = (v_{k-2}, v_{k-1}), e_k = (v_{k-1}, v_k))$ such that all the edges are distinct. An Euler trail traverses every edge of G.

a. Show that G has an Euler trail if and only if it has at most two vertices of odd degree.

b. Use your answers to Exercise 5.22 to develop an efficient parallel algorithm to identify an Euler trail whenever it exists.

5.24. *The definition of a **walk** is similar to that of a trail (given in Exercise 5.23), except we relax the condition that all the edges in the sequence

must be distinct. A **tour** is a closed walk such that every edge appears in the walk.

The **Chinese postman problem** is the following. Given a connected weighted graph G, determine a minimum-weight tour of G.

a. Suppose that G has no vertices of odd degree. Develop an $O(\log n)$ time parallel algorithm to solve the Chinese postman problem. State the total number of operations used.

b. Suppose that G has exactly two vertices of odd degree. Develop an $O(\log^2 n)$ time algorithm to solve the Chinese postman problem. *Hint:* Make use of the algorithm to determine the Euler tour of a graph outlined in Exercise 5.22.

5.25. Given an undirected connected graph $G = (V, E)$, the **strong-orientation problem** is to assign a direction to each edge so that the resulting directed graph is strongly connected; that is, for any given pair u and v of vertices, there exist directed paths from u to v and from v to u.

a. Prove that, if G is bridgeless, then G has a strong orientation.

b. Use the ear-decomposition algorithm (Algorithm 5.6) to obtain an efficient parallel algorithm for the strong-orientation problem.

5.26. Prove Theorem 5.5.

5.27. The ear-decomposition algorithm (Algorithm 5.6) generates each ear as a set of edges. Develop a parallel algorithm to produce, for each ear, the ordered sequence of edges representing a simple path. Your algorithm should not asymptotically increase the resource bounds required by Algorithm 5.6.

5.28. *Given a biconnected graph, you are asked to determine an *open* ear decomposition. The algorithm (Algorithm 5.6) given in the text does not necessarily generate an open ear decomposition, but it can be refined to do so. Develop such an algorithm without increasing the resource bounds. *Hint:* Refine the labeling procedure such that, for each nontree edge (u, v) with $x = lca(u, v)$, the two edges incident on x and part of the cycle induced by (u, v) do not have the same label as that of (u, v).

5.29. A **topological sort** can be defined as follows. Given a directed acyclic graph $G = (V, E)$ with n vertices, assign labels l to V from $\{1, 2, \ldots, n\}$ such that $l(u) < l(v)$ whenever $\langle u, v\rangle \in E$. Develop a parallel algorithm to perform a topological sort on G. What are the corresponding resource bounds used by your algorithm?

5.30. Let $G = (V, E)$ be an undirected connected graph with n vertices. Show how to identify a breadth-first search tree of G in $O(\log^2 n)$ time. What is the total number of operations used?

5.31. Let $G = (V, E)$ be a directed graph with $|V| = n$. Let R be the relation uRv if and only if either $u = v$, or, u is reachable from v and v is

reachable from u. We say that a vertex x is *reachable* from a vertex y if there is a directed path from y to x.

a. Show that R is an equivalence relation on V. The induced equivalence classes are called the **strongly connected components** of G.

b. Show how to identify the strongly connected components in $O(\log^2 n)$ time, using a total of $O(M(n) \log n)$ operations, where $M(n)$ is the best known sequential bound for multiplying two $n \times n$ matrices.

5.32. Let $G = (V, E)$ be a directed acyclic graph such that $V = \{u_1, u_1', u_2, u_2', \ldots, u_n, u_n'\}$, and, whenever $\langle x, y\rangle \in E$, $\langle x', y'\rangle \in E$ (by convention, $(u_i')' = u_i$). Assign labels l to all the vertices in V from $\{0, 1\}$ such that $l(u_i) \neq l(u_i')$ and no arc $\langle x, y\rangle \in E$ satisfies $l(x) = 1$ and $l(y) = 0$. What are the corresponding resource bounds of your algorithm?

5.33. Let F be a 2-CNF formula; that is, $F = C_1 \wedge C_2 \wedge \ldots \wedge C_m$, where each $C_i = u_{i1} \vee u_{i2}$, u_{i1} and u_{i2} are literals from n Boolean variables $X = \{x_1, \ldots, x_n\}$. The formula F is *satisfiable* if there exists a truth assignment $t : X \rightarrow \{0, 1\}$ that satisfies F. We use u' to denote the complement of u.

Let $G = (V, E)$ be a directed graph constructed from F as follows. The vertex set V is given by $V = \{x_1, \ldots, x_n, x_1', \ldots, x_n'\}$, and, for every $C_i = u_{i1} \vee u_{i2}$, we insert two arcs $\langle u_{i1}', u_{i2}\rangle$ and $\langle u_{i2}', u_{i1}\rangle$.

a. Show that, if F is satisfiable, then all the literals appearing in a strongly connected component of G must be assigned the same truth value.

b. Develop a parallel algorithm to determine a truth assignment for F, whenever F is satisfiable. *Hint:* Identify each strongly connected component with a single supervertex, and use your answer to Exercise 5.32.

5.34. *Let $G = (V, E)$ be a directed acyclic graph such that, for every two vertices u and v, there is at most one path from u to v. Develop an $O(\log n)$ time CREW PRAM algorithm to compute the transitive closure of G, where $|V| = n$.

5.35. Let $G = (V, A)$ be a weighted directed graph that is represented by its weight matrix W. Assume that G has no negative weight cycles. Show how to compute the distance matrix D representing the weights of all pairs shortest paths in $O(\log n)$ time. What is the total number of operations used?

5.36. Let $G = (V, A)$ be a weighted graph such that the weight of each arc is nonnegative. Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j using vertices from $\{1, 2, \ldots, k\}$.

a. Show that $d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$, for all $k \geq 1$.

b. Based on the relation established in part (a), develop an algorithm to solve the all pairs shortest-paths problem in $O(n^3)$ sequential time. The resulting algorithm is called the **Floyd–Warshall algorithm.**

c. Discuss the parallel implementation of the Floyd-Warshall algorithm on a CREW PRAM with p processors, where $1 \leq p \leq n$. Compare it to the algorithm presented in Section 5.5.3.

5.37. Let $G = (V, E)$ be a directed graph. An ear decomposition of G can be defined as in the case of undirected graphs, except the paths and the cycles have to be directed. Suppose that G is strongly connected. Develop an efficient parallel algorithm to determine an ear decomposition of G.

5.38. * Given a set $S = \{1, 2, \ldots, n\}$, an initial partition of S, $B = \{B_1, \ldots, B_m\}$, and a function $f: S \rightarrow S$, we want to find the equivalence classes under the relation $\Delta : i \Delta j$ if and only if $f^{(k)}(i)$ and $f^{(k)}(j)$ are in the same partition for all k, where $f^{(k)}$ is the function composition of f (k times). Develop an efficient parallel algorithm to determine the equivalence classes. This problem is called the **coarsest-partitioning problem.** *Hint:* The graph of f defines a pseudoforest. Give a characterization of the equivalence classes corresponding to each tree. Reduce each tree, and try to relate trees.

Bibliographic Notes

The basic strategy behind our connected-components algorithm for dense graphs (Algorithm 5.1) was described by Hirschberg [15] and by Hirschberg et al. [16]. The total amount of work was improved from $O(n^2 \log n)$ to $O(n^2)$ in [8], with preservation of the CREW PRAM model. An EREW version has appeared in [28], and a common CRCW version, similar to the one covered in the text, was described in [40]. Extensions of the CREW PRAM algorithm that work well for sparse graphs were reported in [14, 22, 42]. The $O(\log n)$-time CRCW PRAM algorithm to determine the connected components for sparse graphs (Algorithm 5.2) was described by Shiloach and Vishkin [36]; it was later simplified in [5]. An involved version that uses asymptotically fewer operations appeared in [9]. The three basic MST algorithms mentioned at the beginning of Section 5.2 are named after some of their inventors; Kruskal's algorithm appeared in [23], Prim's algorithm in [31], and Sollin's algorithm in [6]. For the historical origins of these algorithms, the reader may consult [38]. There are several more involved sequential algorithms that are more suitable for sparse graphs [7, 11, 43]. The first parallel implementation of Sollin's algorithm was described by Savage [34] and JáJá [17]; see also [35]. The total amount of work for that algorithm was later improved by a logarithmic factor in [8]. For the parallel implementation of other MST algorithms, see [25]. The biconnected-components algorithm (Algorithm 5.5) presented in the text was developed by Tarjan and Vishkin [37]; another parallel algorithm suitable for dense graphs appeared in [39]. The ear-decomposition algorithm was described by Maon et al. [26] and by Miller and Ramachandran [27]. Parallel implementations of the transitive-closure algorithm were presented by Hirschberg [15] and by Reghbati and Corneil [32]. Parallel implementations of the all pairs shortest paths appeared in [34] and [24]. Concerning the matrix-multiplication problem, the best known algorithm uses $O(n^{2.376})$ arithmetic operations; it appeared in [10]. The relationships among matrix multiplication, transitive closure, and shortest paths are described in [1]. Other important graph problems not covered in this

chapter include network flows [13], planarity testing [19], [20], [29], triconnectivity [30], and chordal graphs [21].

The improved connected-components algorithm for sparse graphs sketched in Exercise 5.11 is derived from the work of [9]. A solution to Exercise 5.13 can be found in [5]. The problem of coloring the edges of a bipartite graph was addressed in [12], where a solution to Exercise 5.21 can be found. Determining Euler circuits for directed graphs was addressed in [2, 4]. The algorithm sketch given in Exercise 5.22 is from [4]. Parallel algorithms for the strong-orientation problem defined in Exercise 5.25 have appeared in [3, 41]. Early work on parallel algorithms for determining the strongly connected components of a directed graph, as defined in Exercise 5.31, was reported in [32, 17]. Exercise 5.34 appears as a subproblem of the problem of parsing context-free languages addressed in [33]. A parallel algorithm to handle the coarsest-partitioning algorithm of Exercise 5.38 appeared in [18].

References

1. Aho, A. V., J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.
2. Atallah, M., and U. Vishkin. Finding Euler tours in parallel. *Journal of Computer and System Sciences*, 29(30):330-337, 1984.
3. Atallah, M. J. Parallel strong orientation of an undirected graph. *Information Processing Letters*, 18(1):37-39, 1984.
4. Awerbuch, B., A. Israeli, and Y. Shiloach. Finding Euler circuits in logarithmic parallel time. In *Proceedings 1984 Symposium on Theory of Computing*, Washington, DC, 1984, pp. 249-257.
5. Awerbuch, B., and Y. Shiloach. New connectivity and MSF algorithms for ultracomputer and PRAM. *IEEE Transactions on Computers*, 36(10):1258-1263, 1987.
6. Berge, C., and A. Ghouila-Houri. *Programming, Games, and Transportation Networks.* John Wiley, New York, 1965.
7. Cheriton, D., and R. E. Tarjan. Finding minimum spanning trees. *SIAM J. Computing*, 5(4):724-742, 1976.
8. Chin, F. Y., J. Lam, and I. Chen. Efficient parallel algorithms for some graph problems. *Communications of the ACM*, 25(9):659-665, 1982.
9. Cole, R., and U. Vishkin. Approximate parallel scheduling. Part II: Applications to optimal parallel graph algorithms in logarithmic time. *Information and Computation*, 92(1):1-47, 1991.
10. Coppersmith, D., and S. Winograd. Matrix multiplication via arithmetic progressions. *J. of Symbolic Computations*, 9(3):251-280, 1990.
11. Fredman, M. L., and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *JACM*, 34(3):209-221, 1987.
12. Gibbons, A., and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, Cambridge, Great Britain, 1988.
13. Goldberg, A. Parallel algorithms for network flow problems, in *Synthesis of Parallel Algorithms*, J. H. Reif ed. Morgan Kaufmann, San Mateo, CA, 1991.

14. Han, Y., and R. A. Wagner. An efficient and fast parallel-connected component algorithm. *JACM*, 37(3):626–642, 1990.

15. Hirschberg, D. S. Parallel algorithms for the transitive closure and the connected components problems. In *Proceedings Eighth Annual ACM Symposium on Theory of Computing*, Hershey, PA, 1976. ACM Press, New York, pp. 55–57.

16. Hirschberg, D. S., A. K. Chandra, and D. V. Sarwate. Computing connected components on parallel computers. *Communications of the ACM*, 22(8):461–464, 1979.

17. JáJá, J. Graph connectivity problems on parallel computers. Technical Report CS-78-05, Department of Computer Science, Pennsylvania State University, University Park, PA, 1978.

18. JáJá, J, and S. Rao Kosaraju. Parallel algorithms for planar graph isomorphism and related problems. *IEEE CAS Transactions*, 35(3):304–311, 1988.

19. JáJá, J., and J. Simon. Parallel algorithms in graph theory: planarity testing. *SIAM J. Computing*, 11(2):314–328, 1982.

20. Klein, P., and J. H. Reif. An efficient parallel algorithm for planarity. *JCSS*, 37(2):190–246, 1988.

21. Klein, P. Parallel algorithms for chordal graphs, in *Synthesis of Parallel Algorithms*, J. H. Reif, ed. Morgan Kaufman, San Mateo, CA, 1991.

22. Kruskal, C. P., L. Rudolph, and M. Snir. Efficient parallel algorithms for graph problems. In *Proceedings 1986 International Conference on Parallel Processing*, St. Charles, IL, 1986, pp. 278–284.

23. Kruskal, J. B. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Amer. Math. Soc.*, 7(1):48–50, 1956.

24. Kucera, L. Parallel computation and conflicts in memory access. *Information Processing Letters*, 14(2):93–96, 1982.

25. Kwan, S. C., and W. L. Ruzzo. Adaptive parallel algorithms for finding minimum spanning trees. In *Proceedings 1984 International Parallel Processing Conference*, St. Charles, IL, 1984, pp. 439–443.

26. Maon, Y., B. Schieber, and U. Vishkin. Parallel ear decomposition search (eds) and st-numbering in graphs. *Theoretical Computer Science*, 47(3):277–296, 1986.

27. Miller, G. L., and V. Ramachandran. Efficient parallel ear decomposition with applications. Unpublished manuscript, Mathematical Sciences Research Institute, Berkeley, CA, 1986.

28. Nath, D., and S. N. Maheshwari. Parallel algorithms for the connected components and minimal spanning trees. *Information Processing Letters*, 14(1):7–11, 1982.

29. Ramachandran, V., and J. Reif. Planarity testing in parallel. Technical report TR-90-15, Department of Computer Sciences, University of Texas at Austin, Austin, Texas, 1990.

30. Ramachandran, V. Parallel open ear decomposition with applications to graph biconnectivity and triconnectivity, in *Synthesis of Parallel Algorithms*, J. H. Reif, ed. Morgan Kaufman, San Mateo, CA, 1991.

31. Prim, R. C. Shortest connection networks and some generalizations. *Bell System Tech. J.*, 36(6):1389–1401, 1957.

32. Reghbati, E., and D. G. Corneil. Parallel computations in graph theory. *SIAM J. Computing*, 7(2):230–237, 197.

33. Rytter, W. On the complexity of parallel parsing of general context-free languages. *Theoretical Computer Science*, 47(3):315–322, 1987.

34. Savage, C. Parallel Algorithms for Graph Theoretic Problems. PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1978.

35. Savage, C., and J. JáJá. Fast, efficient parallel algorithms for some graph problems. *SIAM J. Computing*, 10(4):682–691, 1981.

36. Shiloach, Y., and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.

37. Tarjan, R. E., and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. *SIAM J. Computing*, 14(4):862–874, 1985.

38. Tarjan, R. E. *Data Structures and Network Algorithms* SIAM, Philadelphia, PA, 1983.

39. Tsin, Y. H., and F. Y. Chin. Efficient parallel algorithms for a class of graph theoretic problems. *SIAM J. Computing*, 13(3):580–599, 1984.

40. Vishkin, U. An optimal parallel connectivity algorithm. *Discrete Applied Mathematics*, 9(2):197–207, 1984.

41. Vishkin, U. On efficient parallel strong orientation. *Information Processing Letters*, 20(5):235–240, 1985.

42. Wyllie, J. C. *The Complexity of Parallel Computations.* PhD thesis, Computer Science Department, Cornell University, Ithaca, NY, 1979.

43. Yao, A. An $O(|E| \log \log |V|)$ algorithm for finding minimum spanning trees. *Information Processing Letters*, 4(1):21–23, 1975.

6

Planar Geometry

Computational geometry is the study of designing efficient algorithms to handle computational problems dealing with collections of objects in Euclidean space. This rich class of problems arises in many applications, such as computer graphics, computer-aided design (CAD), robotics, pattern recognition, and statistics. Our goal in this chapter is to present efficient parallel algorithms for a few selected geometric problems in the plane.

Three main techniques will dominate this chapter. The first is the **divide-and-conquer** technique, which has already provided a powerful paradigm for sequential computational geometry. The second technique is an elegant data structure, called the **plane-sweep tree,** which can be used for the parallel implementation of the **plane-sweeping** paradigm. The third technique is the **pipelined divide-and-conquer** used earlier in Chapter 4 to develop the optimal $O(\log n)$ time sorting algorithm.

We start in Section 6.1 by refining the divide-and-conquer algorithm presented in Chapter 2 for computing the *convex hull* of a set of points. The resulting algorithm runs in $O(\log n)$ time using a total of $O(n \log n)$ operations. This algorithm is then used in Section 6.2 to determine the *intersection of a set of half-planes* within the same asymptotic bounds. The plane-sweeping method and the plane-sweep tree for a set of horizontal segments are introduced in Section 6.3, and are applied to the problem of determining,

for each endpoint of a given set of horizontal segments, the closest segment directly below it. The next two sections are devoted to a *visibility problem* and to the *dominance-counting problem*, both of which can be solved efficiently by an extension of the pipelined divide-and-conquer strategy.

For computational-geometry algorithms, it is useful to expand the power of the PRAM model. Since we will be dealing with real numbers representing the coordinates of a point in the plane with respect to a coordinate system, we will allow our PRAM model to perform a **real** arithmetic operation (such as $+$, $-$, $\times$, $/$) or a **real** comparison operation, at unit cost. We will assume that each real number can be stored in a single memory location. Occasionally, we will allow additional primitive operations, such as computing the square root of a positive real number.

6.1 The Convex-Hull Problem Revisited

We have already encountered the convex-hull problem in Section 2.3 in the context of introducing the divide-and-conquer strategy. A simple algorithm (Algorithm 2.6) was presented that computes the convex hull of a set of n points in $O(\log^2 n)$ time using a total of $O(n \log n)$ operations. In this section, we refine this algorithm such that the merge operation, required for combining the two solutions of the subproblems created by the divide-and-conquer strategy, can be performed in $O(1)$ time using a linear number of operations. It turns out that the parallel-search algorithm presented in Section 4.1 will play a crucial role in the development of such a merge procedure.

6.1.1 DEFINITIONS

We start by presenting several definitions, some of which we already introduced in Section 2.3.

Let $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ be two points of the Euclidean plane. The **segment** $s = p_1 p_2$ is defined to be the set of points q satisfying $q = \alpha p_1 + (1 - \alpha)p_2$, for all α such that $0 \leq \alpha \leq 1$. That is, s consists of all the points lying on the straight-line segment joining p_1 and p_2, including p_1 and p_2. We call p_1 and p_2 the **endpoints** of s.

Given an arbitrary point p in the plane, we use the notation $x(p)$ and $y(p)$ to denote the values of p's x and y coordinates, respectively. Let L be a line specified by the equation $y = ax + b$. A point $p = (\alpha, \beta)$ is **below** L (or L is **above** p) if $\beta < a\alpha + b$. We can, in a similar fashion, define p **above** L or L **below** p.

A **polygonal chain** is an ordered set of segments $P = (s_0 = p_0p_1, s_1 = p_1p_2, \ldots, s_{n-1} = p_{n-1}p_n)$. A polygonal chain is **simple** if each segment s_i intersects only s_{i-1} and s_{i+1} and then only at the endpoints p_i and p_{i+1}, respectively, whenever s_{i-1} or s_{i+1} exists. For the remainder of this chapter, we shall assume that all polygonal chains are simple. If $p_0 = p_n$ in a polygonal chain P, then P defines two regions: the bounded region inside the chain, which is called a **polygon** whose **boundary** is the polygonal chain P, and the unbounded region that lies outside P.

A polygon Q is **convex** if, given any two points p and q in Q, the segment $s = p_1p_2$ lies entirely in Q. A **tangent** or a **supporting line** of a convex polygon Q is a line L passing through a vertex of Q such that Q lies entirely on one side of L.

EXAMPLE 6.1:

Figure 6.1(a) shows a nonsimple polygonal chain. The polygon in Fig. 6.1(b) is nonconvex, whereas the polygon in Fig. 6.1(c) is convex, and is shown with a tangent L (dotted line). □

Let $S = (p_1, p_2, \ldots, p_n)$ be a set of n points in the plane. The **convex hull** of S is the smallest convex polygon containing all the n points of S. The convex-hull problem is to determine an ordered (say, clockwise) list $CH(S)$ of the points of S defining the boundary of the convex hull of S. Each point of $CH(S)$ is called an **extreme point** of S or a **vertex** of $CH(S)$.

For simplicity, we assume that no two points of S have the same x or y coordinate. Let p and q be the points of S with the *smallest* and the *largest* x coordinate, respectively. Clearly, the points p and q belong to $CH(S)$; they

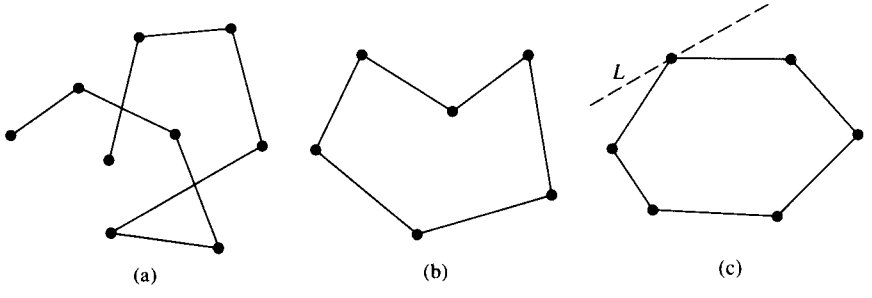


FIGURE 6.1
Polygonal chains and polygons. (a) A nonsimple polygonal chain. (b) A nonconvex polygon. (c) A convex polygon and a tangent L.

partition $CH(S)$ into an **upper hull** $UH(S)$, consisting of all points from p to q of $CH(S)$ (clockwise), and a **lower hull** $LH(S)$, defined similarly from q to p.

EXAMPLE 6.2:

Consider the set S of points shown in Fig. 6.2. In this case, the convex hull is given by CH(S) = $(v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_1)$, and the upper and the lower hulls are $UH(S) = (v_1, v_2, v_3, v_4, v_5, v_6)$ and $LH(S) = (v_6, v_7, v_8, v_9, v_{10}, v_1)$. □

6.1.2 A DIVIDE-AND-CONQUER STRATEGY

As we saw in Section 2.3, the convex-hull problem can be solved by a divide-and-conquer strategy. Let $S = (p_1, p_2, \ldots, p_n)$, where n is assumed to be a power of 2. We start by sorting the p_i's by their x coordinates. Assume for the remainder of this section that the list of points defining S satisfies $x(p_1) < x(p_2) < \ldots < x(p_n)$, where $x(p)$ is the x coordinate of p. The parallel algorithm developed in Section 2.3 consists of recursively determining $CH(S_1)$ and $CH(S_2)$, where $S_1 = (p_1, p_2, \ldots, p_{n/2})$ and $S_2 = (p_{(n/2)+1}, p_{(n/2)+2}, \ldots, p_n)$. Then, using the upper and the lower common tangents of $CH(S_1)$ and $CH(S_2)$, $CH(S_1)$ and $CH(S_2)$ are combined to obtain $CH(S)$.

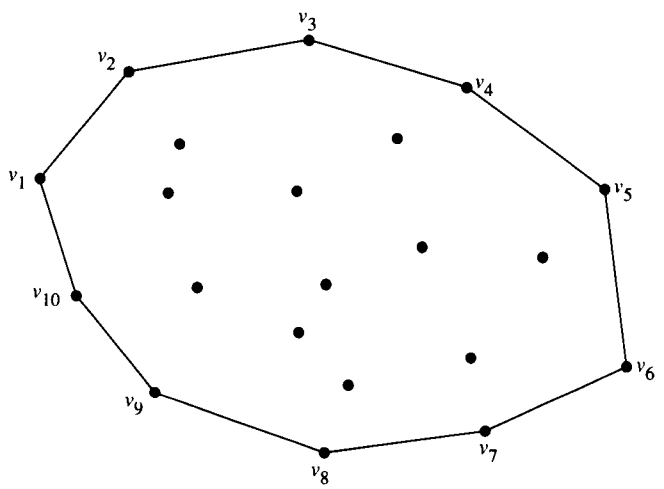


FIGURE 6.2
The convex hull of a set of points. The upper hull is given by $(v_1, v_2, v_3, v_4, v_5, v_6)$; the lower hull is given by $(v_6, v_7, v_8, v_9, v_{10}, v_1)$.

The analysis presented in Section 2.3 assumed the existence of an $O(\log n)$ time sequential algorithm to determine the upper or the lower common tangent. We present next an $O(1)$ time parallel algorithm to compute the upper or lower common tangent of $CH(S_1)$ and $CH(S_2)$, using a linear number of operations.

6.1.3 A CONSTANT-TIME ALGORITHM FOR COMPUTING THE UPPER COMMON TANGENT

Let $S = (p_1, p_2, \ldots, p_n)$ be the given set of points such that $x(p_1) < x(p_2) < \cdots < x(p_n)$, and let $S_1 = (p_1, p_2, \ldots, p_{n/2})$ and $S_2 = (p_{(n/2)+1}, p_{(n/2)+2}, \ldots, p_n)$. Recall that the **upper common tangent** between $CH(S_1)$ and $CH(S_2)$ is the common tangent such that $CH(S_1)$ and $CH(S_2)$ are below it (see Fig. 6.3.) We can define the lower common tangent in a similar fashion. In the remainder of this section, we develop a constant-time algorithm to determine the upper common tangent. We can find the lower common tangent by using a similar algorithm.

Let $UH(S_1) = (r_1, \ldots, r_s)$ and $UH(S_2) = (q_1, \ldots, q_t)$ be the upper hulls of S_1 and S_2, respectively, given in a left-to-right order, where $r_i, q_j \in \{p_1, \ldots, p_n\}$ for $1 \leq i \leq s$ and $1 \leq j \leq t$. To combine $UH(S_1)$ and $UH(S_2)$, we need to determine points $u = r_i$ and $v = q_j$ defining the upper common tangent T. Once we have determined them, the upper hull $UH(S)$ is the array consisting of the first i entries of $UH(S_1)$ and the last $t - j + 1$ entries of

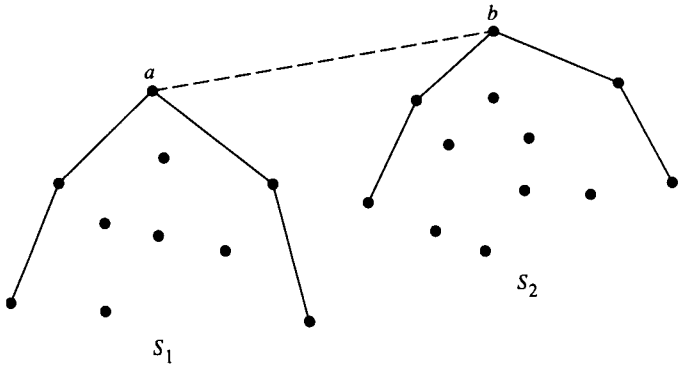


FIGURE 6.3
The line (a, b), which is the upper common tangent of the upper hulls of S_1 and S_2.

$UH(S_2)$. If s and t are given as part of the input, $UH(S)$ (and its size) can then be determined in $O(1)$ time, using a total of $O(n)$ operations from the indices i and j defining u and v.

Let us first discuss the problem of determining the tangent between a point r_i of $UH(S_1)$ and $UH(S_2)$; that is, we wish to find $q_{j(i)}$ such that $UH(S_2)$ is below the line determined by r_i and $q_{j(i)}$. The following observation will make clear how we can accomplish this task by using the parallel-search algorithm.

Lemma 6.1: *Let r_i be an arbitrary point of $UH(S_1)$. Then, given any point q_l of $UH(S_2)$, we can determine in $O(1)$ sequential time whether $q_{j(i)}$ is to the right of q_l, is equal to q_l, or is to the left of q_l, where $q_{j(i)}$ is the point of $UH(S_2)$ such that $r_i q_{j(i)}$ is the tangent to $UH(S_2)$.*

Proof: Let L be the line determined by r_i and q_l, and let L' (respectively L'') be the portion of L strictly to the left (respectively, right) of q_l (see Fig. 6.4.) Then, if L' is above the segment $q_{l-1}q_l$ and L'' is below the segment $q_l q_{l+1}$, then $q_{j(i)}$ is to the right of q_l. If $q_l = q_{j(i)}$, then q_{l-1} and q_{l+1} are below L. Otherwise, $q_{j(i)}$ is to the left of q_l.

Once r_i and q_l are given, we can determine each of the three conditions in $O(1)$ sequential time. Therefore, the lemma follows. □

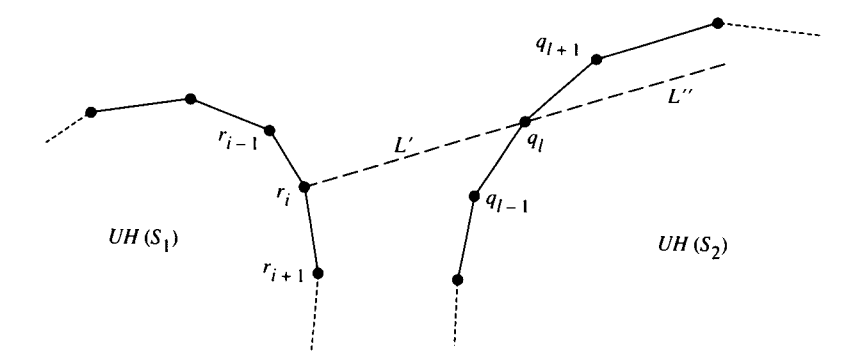


FIGURE 6.4
Determination of the tangent from r_i to $UH(S_2)$. In this case, L' is above $q_{l-1}q_l$, and L'' is below $q_l q_{l+1}$. Hence, the tangent from r_i to $UH(S_2)$ touches a point of $UH(S_2)$ that is to the right of q_l.

Corollary 6.1: *Given the two upper hulls $UH(S_1)$, $UH(S_2)$, and a point r_i of $UH(S_1)$, the tangent $r_i q_{j(i)}$ to $UH(S_2)$ can be determined in $O\left(\frac{\log t}{\log k}\right)$ time using k processors, where t is the number of points defining $UH(S_2)$ and $1 < k \leq t$.*

Proof: The array $UH(S_2) = (q_1, \ldots, q_t)$ is given in sorted order (clockwise) from left to right. We can then use the parallel-search algorithm (Algorithm 4.1) to find $q_{j(i)}$ as follows. We choose k points of $UH(S_2)$ that split $UH(S_2)$ into $k + 1$ portions, each with approximately the same number of points. Lemma 6.1 can be used to determine, in $O(1)$ time, the portion containing $q_{j(i)}$ using k processors. We repeat the process until $q_{j(i)}$ is identified. The analysis is identical to that of Algorithm 4.1. □

Another key observation is given in the next lemma.

Lemma 6.2: *Let (u, v) be the upper common tangent of the two upper hulls $UH(S_1)$ and $UH(S_2)$. Suppose that, for an arbitrary point r_i of $UH(S_1)$, we are given $q_{j(i)}$ such that the line segment $r_i q_{j(i)}$ defines the tangent to $UH(S_2)$. Then, in $O(1)$ sequential time, we can determine whether u is to the left of, is equal to, or is to the right of r_i.*

Proof: Note that, if $r_i\, q_{j(i)}$ is tangent to $UH(S_1)$, then u must be equal to r_i, since $r_i q_{j(i)}$ is already tangent to $UH(S_2)$. Otherwise, u is to the left of r_i if and only if r_{i-1} is above $r_i\, q_{j(i)}$ (see Fig. 6.5.) □

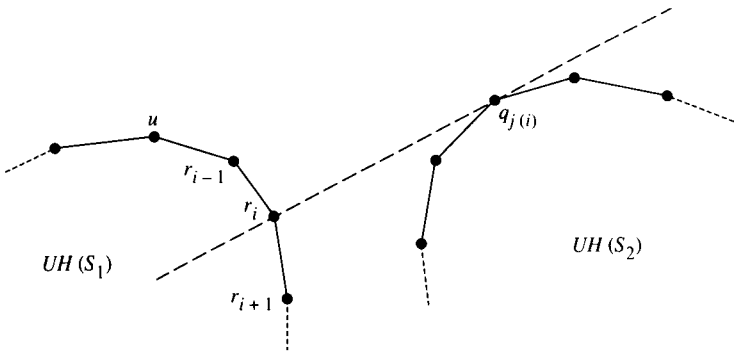


FIGURE 6.5
Determination of the location of the point u where the upper common tangent touches $UH(S_1)$, given the tangent $r_i q_{j(i)}$ to $UH(S_2)$. In the case shown in the picture, r_{i-1} is above the tangent, and hence u must be to the left of r_i.

Lemmas 6.1 and 6.2 allow us to determine, for any given point r_i of $UH(S_1)$, whether the point u of the upper common tangent (u, v) appears to the left of, to the right of, or equal to r_i in $O\left(\frac{\log t}{\log k}\right)$ time using k processors. If we take k to be approximately $\sqrt{t}$, we obtain an $O(1)$ time parallel algorithm that determines, for any given point r_i, whether u is equal to, is to the left of, or is to the right of r_i.

These observations lead to a parallel-search algorithm to isolate the point u as follows. We choose $\sqrt{s}$ points of $UH(S_1)$, which divide $UH(S_1)$ into almost-equal portions. We can then isolate u within a single portion in $O(1)$ time using $\sqrt{s}\sqrt{t}$ processors. Similarly, we can isolate v within a portion of $UH(S_2)$ containing approximately $\sqrt{t}$ points in $O(1)$ time using $\sqrt{s}\sqrt{t}$ processors. The complexity bounds can be stated as $O(1)$ time, with a total of $O(\sqrt{s}\sqrt{t}) = O(n)$ operations.

We now have at most $\sqrt{s}$ candidates for the point u and $\sqrt{t}$ candidates for the point v. For each pair of candidates, we can check, in $O(1)$ sequential time, whether that pair constitutes the upper common tangent between $UH(S_1)$ and $UH(S_2)$. Therefore, we can test all pairs concurrently; hence, we can identify the unique upper common tangent in $O(1)$ time, using $O(n)$ operations.

We are now ready to state the algorithm more formally.

ALGORITHM 6.1
(Upper Common Tangent)

Input: *The upper hulls* $UH(S_1) = (r_1, r_2, \ldots, r_s)$ *and* $UH(S_2) = (q_1, q_2, \ldots, q_t)$ *in sorted order from left to right, where* $S_1 = (p_1, p_2, \ldots, p_{n/2})$ *and* $S_2 = (p_{(n/2)+1}, p_{(n/2)+2}, \ldots, p_n)$ *such that* $x(p_1) < x(p_2) < \cdots < x(p_n)$. *Assume that* $\sqrt{s}$ *and* $\sqrt{t}$ *are integers.*
Output: *Points u and v such that the line determined by u and v is the upper common tangent between* $UH(S_1)$ *and* $UH(S_2)$.
begin
1. *For each i such that* $1 \le i \le \sqrt{s}$, *find* $q_{j(i\sqrt{s})}$ *such that* $r_{i\sqrt{s}}$ $q_{j(i\sqrt{s})}$ *is the tangent to* $UH(S_2)$.
2. *For each i such that* $1 \le i \le \sqrt{s}$, *determine whether u is to the left of, is equal to, or is to the right of* $r_{i\sqrt{s}}$. *If* $u = r_{i\sqrt{s}}$, *for some i, then we are done. Otherwise, deduce the block* $A = (r_{l\sqrt{s}+1}, \ldots, r_{(l+1)\sqrt{s}-1})$ *containing u.*
3. *For each* r_i *in block A, determime* $q_{j(i)}$ *such that* $r_i q_{j(i)}$ *is the tangent to* $UH(S_2)$, *set u:* $= r_i$ *and v:* $= q_{j(i)}$ *if* $r_i q_{j(i)}$ *is also tangent to* $UH(S_1)$.

end

Theorem 6.1: *Algorithm 6.1 correctly computes the upper common tangent of the two upper hulls $UH(S_1)$ and $UH(S_2)$. This algorithm runs in $O(1)$ time, using a linear number of operations.*

Proof: The correctness proof follows essentially from Lemmas 6.1 and 6.2 and the discussion presented just before the statement of the algorithm.

We estimate the running time assuming that there are $s + t$ processors available. Step 1 can be executed in $O(1)$ using $\sqrt{t}$ processors for each i (Lemma 6.1 and its corollary.) Since $\sqrt{st} \leq s + t$, all the $q_{j(i\sqrt{s})}$'s can be computed in $O(1)$ time, using $s + t$ processors. Step 2 can be done in $O(1)$ time with $O(\sqrt{s})$ processors. Similar arguments hold for step 3. Therefore, the whole algorithm can be executed in $O(1)$ time, using $s + t$ processors. Hence, the corresponding total number of operations is $O(s + t) = O(n)$. □

6.1.4 PUTTING THE PIECES TOGETHER

Combining the divide-and-conquer strategy with our fast parallel algorithm to compute the upper or lower common tangent, we obtain the following theorem.

Theorem 6.2: *The convex hull of a set of n points in the plane can be computed in $O(\log n)$ time, using a total of $O(n \log n)$ operations. Hence, Algorithm 6.1 is optimal.*

Proof: Let $S = (p_1, p_2, \ldots, p_n)$ be the given set of n points. We preprocess S by sorting the points according to their x coordinates. This preprocessing step takes $O(\log n)$ time, using $O(n \log n)$ operations if we use the pipelined merge-sort algorithm (Algorithm 4.4) Hence, assume that $x(p_1) < x(p_2) < \cdots < x(p_n)$.

We use a divide-and-conquer strategy to determine $CH(S)$ as follows. If $n \leq 4$, we identify $CH(S)$ by using a brute-force method. Otherwise, we call the algorithm recursively to compute $CH(S_1)$ and $CH(S_2)$, where $S_1 = (p_1, p_2, \ldots, p_{n/2})$ and $S_2 = (p_{(n/2)+1}, p_{(n/2)+2}, \ldots, p_n)$. We then merge $CH(S_1)$ and $CH(S_2)$ by computing the upper and the lower common tangents of $CH(S_1)$ and $CH(S_2)$ and deducing $CH(S)$.

There are $O(\log n)$ iterations, each of which takes $O(1)$ time, using a linear number of operations. Hence, the completion of the divide-and-conquer algorithm requires $O(\log n)$ time and a total of $O(n \log n)$ operations. Therefore, the overall algorithm can be executed within these bounds.

Since the convex-hull problem can be reduced to sorting, the resulting algorithm is optimal. □

PRAM Model: Concurrent-read capability is required by our sorting algorithm and by the parallel-search algorithm. No concurrent-write capability is needed. Hence, our algorithm runs on the CREW PRAM model. □

6.2 Intersections of Convex Sets

An important class of problems in computational geometry is the computation of the intersections of a collection of geometric objects. This class of problems has many applications, such as pattern recognition, VLSI CAD tools, linear programming, and computer graphics. In this section, we restrict our discussion to the problem of determining the boundary of the **intersection of half-planes.** Other related problems—such as determining the intersection of two convex polygons, and finding a separating line between two convex polygons, if it exists—are left as Exercises 6.12 and 6.19.

6.2.1 INTERSECTION OF HALF-PLANES

Consider a line L in the plane defined by the equation $y = ax + b$. The line L divides the plane into two **half-planes** $H^+(L)$ and $H^-(L)$ such that $H^+(L)$ (respectively, $H^-(L)$) consists of all points (α, β) satisfying $\beta \geq a\alpha + b$ (respectively, $\beta \leq a\alpha + b$). More intuitively, $H^+(L)$ is the set of points on or above L and $H^-(L)$ is the set of points on or below L. It is clear that each of $H^+(L)$ and $H^-(L)$ defines a convex region.

Let $\{L_i : y = a_i x + b_i \mid 1 \leq i \leq n\}$ be a set of n lines, and let $H(L_i)$ be one of the two half-planes determined by L_i. The region defined by the intersection of these half-planes is clearly convex, since the intersection of two convex regions is also convex. However, the resulting region may or may not be bounded, as shown by the following example.

EXAMPLE 6.3:

Consider the two examples shown in Fig. 6.6. The arrow indicates the desired half-plane corresponding to each line. In part (a), the region is bounded; in part (b), the corresponding region is unbounded. □

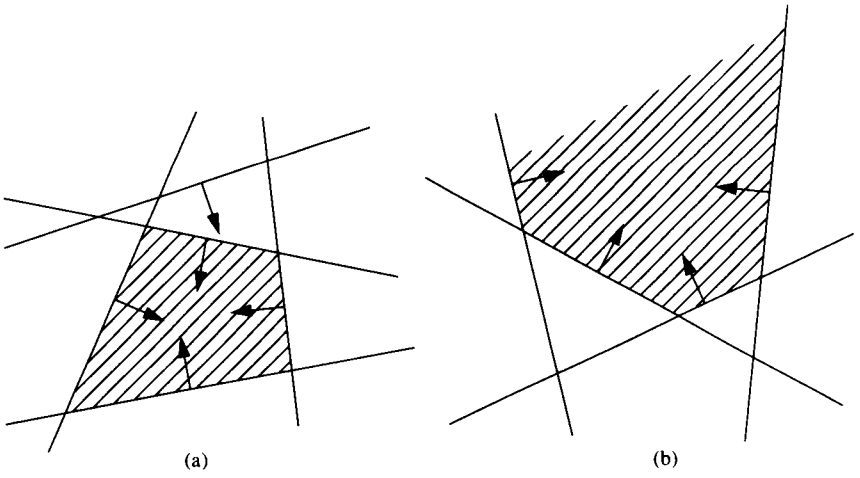


FIGURE 6.6
Intersections of a set of half-planes. (a) Bounded case. (b) Unbounded case.

Our goal is to specify the *polygonal chain* defining the boundary of $\cap H(L_i)$. We next present an $O(\log n)$ time algorithm that relies on using the convex-hull algorithm (Algorithm 6.1) developed in the previous section. The total number of operations used is $O(n \log n)$, which is *optimal* since the problem of sorting n numbers can be reduced to the problem of computing the intersection of n half-planes, as you are asked to show in Exercise 6.11.

6.2.2 REDUCTION TO CONVEX HULL

The problem of determining $\cap H(L_i)$ is somewhat similar to determining the convex hull of a set of points, except that here the convex region (which may be unbounded) is defined by a set of lines, instead of by a set of points. Hence, it is not completely surprising that we can reduce our problem to the convex-hull problem by applying a transformation that maps a line into a point and a point into a line, and that preserves the relative positions. We introduce such a transformation next.

Let T be the transformation that maps a point $p = (a, b)$ into the line $T(p)$ defined by $y = ax + b$, and that maps the line $\{L : y = ax + b\}$ into the point $T(L) = (-a, b)$.

EXAMPLE 6.4:

Consider the point $p = (1, 5)$ and the line $L : y = 2x + 4$. Applying the transformation T, we obtain that $T(p)$ is the line defined by $y = x + 5$, and $T(L)$ is the point $(-2, 4)$, as shown in Fig. 6.7. Note that $T(-2, 4)$ is the line $L' : y = -2x + 4$, which is different from the line L. □

An important fact about the transformation T is stated in the next lemma.

Lemma 6.3: *A point p is below a line L if and only if the line $T(p)$ is below the point $T(L)$.*

Proof: A point $p = (a, b)$ is below the line $\{L : y = cx + d\}$ if and only if $b < ca + d$, which is equivalent to $-ca + b < d$. The point $p = (a, b)$ is mapped into the line $T(p)$ defined by $y = ax + b$, and the line $L : y = cx + d$ is mapped into the point $T(L)$ defined by $(-c, d)$. Hence, the condition $-ca + b < d$ is equivalent to the fact that $T(p)$ is below $T(L)$. Therefore, p is below L if and only if $T(p)$ is below $T(L)$. □

Let $\{L_{i_1}, L_{i_2}, \ldots, L_{i_{n_1}}\}$ be the lines for which $H(L_{i_r}) = H^+(L_{i_r})$, where $1 \leq r \leq n_1$, and let $\{L_{j_1}, L_{j_2}, \ldots, L_{j_{n_2}}\}$ be the remaining lines—that is, $H(L_{j_s}) = H^-(L_{j_s})$, for $1 \leq s \leq n_2$.

Consider the region C^+ defined by $\cap_{1 \leq r \leq n_1} H^+(L_{i_r})$. The region C^+ consists of all the points above all the lines L_{i_r}, for $1 \leq r \leq n_1$. But in the transform domain, $T(C^+) = \{T(p) \mid p \in C^+\}$ consists of all the lines above

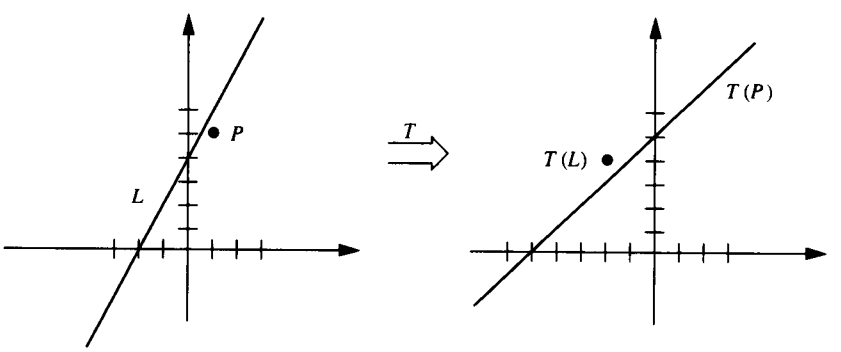


FIGURE 6.7
The transformation T applied to the line L and the point P shown on the left side. The line L is mapped into the point $T(L)$ and the point P is mapped into the line $T(P)$, as shown on the right side.

all the points $T(L_{i_r})$, for $1 \leq r \leq n_1$ (by Lemma 6.3.) Therefore, $T(C^+)$ consists of all the lines above the upper hull determined by the points $T(L_{i_r})$, where $1 \leq r \leq n_1$. The points on this upper hull uniquely define the lines among the L_{i_r}'s that determine the boundary of $\cap_{1 \leq r \leq n_1} H^+(L_{i_r})$.

A line segment of the upper hull joining two consecutive extreme points corresponds to a point in C^+ that is shared by two lines bounding C^+. Therefore, the line segments of the upper hull define the extreme points of C^+, and both sets occur in the same relative order.

More formally, we have the following lemma.

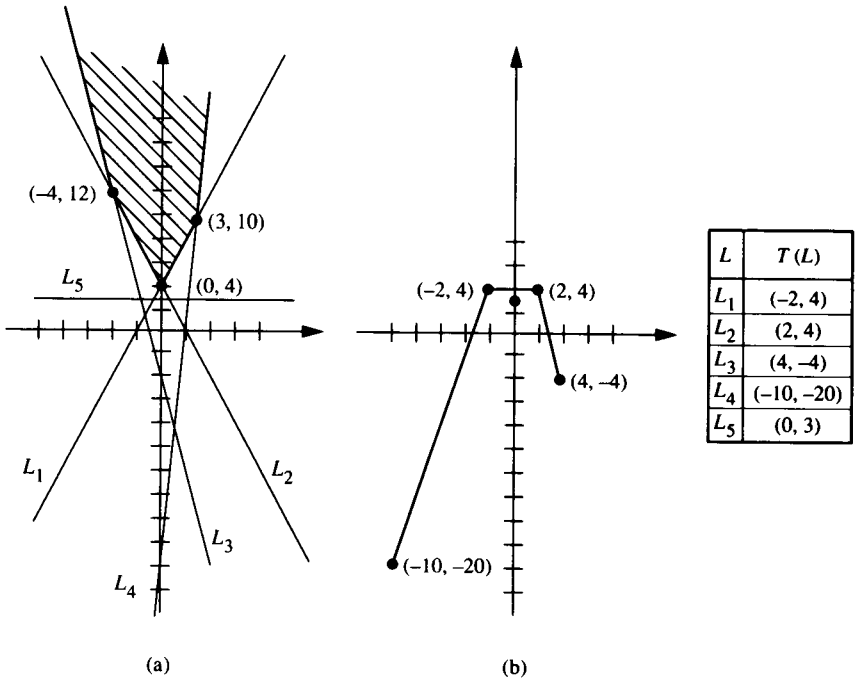
Lemma 6.4: *Let $\mathcal{L} = \{L_{i_1}, L_{i_2}, \ldots, L_{i_{n_1}}\}$ be a given set of lines, and let $C^+ = \cap_{1 \leq r \leq n_1} H^+(L_{i_r})$. Then a line L_{i_j} is on the boundary of C^+ if and only if $T(L_{i_j})$ is an extreme point of the upper hull $UH(T(\mathcal{L}))$, where $T(\mathcal{L})$ is the set of points $\{T(L_{i_1}), T(L_{i_2}), \ldots, T(L_{i_{n_1}})\}$. A point p is an extreme point of C^+ if and only if $T(p)$ is a line joining two consecutive extreme points of $UH(T(\mathcal{L}))$. Moreover, the sequence of lines and points of C^+ corresponding to the sequence of points and lines defining $UH(T(\mathcal{L}))$ forms a polygonal chain that defines the boundary of C^+.* □

EXAMPLE 6.5:

Consider the five lines $L_1 : y = 2x + 4$, $L_2 : y = -2x + 4$, $L_3 : y = -4x - 4$, $L_4 : y = 10x - 20$, and $L_5 : y = 3$. The extreme points of the boundary of $\cap_{1 \leq i \leq 5} H^+(L_i)$ are given by $(-4, 12)$, $(0, 4)$, and $(3, 10)$. The boundary is made up of segments from L_3, L_2, L_1, and L_4, in this order, as shown in Fig. 6.8(a). In the transform domain, we have $T(L_1) = (-2, 4)$, $T(L_2) = (2, 4)$, $T(L_3) = (4, -4)$, $T(L_4) = (-10, -20)$, and $T(L_5) = (0, 3)$. The upper hull of these points is defined by the extreme points $(-10, -20)$, $(-2, 4)$, $(2, 4)$, and $(4, -4)$, as shown in Fig. 6.8(b). Notice that the clockwise ordering of the extreme points of the upper hull is the same as the clockwise ordering of the corresponding lines defining the boundary of $\cap H^+(L_i)$. The equations of the line segments joining successive points of the upper hull are $y = 3x + 10$, $y = 4$, and $y = -4x + 12$. The points corresponding to these lines are precisely the points $(3, 10)$, $(0, 4)$, and $(-4, 12)$ defining the extreme points of $\cap_{1 \leq i \leq 5} H^+(L_i)$ with the same relative ordering. □

We now consider the problem of determining the region C^- defined by $\cap_{1 \leq s \leq n_2} H^-(L_{j_s})$. As before, the region C^- is uniquely specified by the lines corresponding to the points defining the lower hull of the points $T(L_{j_s})$, where $1 \leq s \leq n_2$. The extreme points of C^- are uniquely defined by the line segments of the lower hull.

In summary, the problem of determining the intersection of the set of half-planes $H(L_i)$ comes down to computing the polygonal chain defining the



L	$T(L)$
L_1	(–2, 4)
L_2	(2, 4)
L_3	(4, –4)
L_4	(–10, –20)
L_5	(0, 3)

FIGURE 6.8
The five lines for Example 6.5. (a) Intersections of a set of upper half-planes determined by the lines L_1, L_2, L_3, L_4, and L_5. (b) The transformation T maps the five lines in (a) into the five points shown in (b), as indicated by the accompanying table; the ordered upper hull of these points uniquely define the polygonal chain of the region formed in (a).

boundary of the intersection $C^+ \cap C^-$, where C^+ is the region defined by the intersection of the upper half-planes and C^- is the region defined by the intersection of the lower half-planes. Using the convex-hull algorithm (Algorithm 6.1), we can generate the polygonal chains defining the boundaries of C^+ and C^-.

To compute $C^+ \cap C^-$, we can merge the two sorted lists of the x coordinates of the extreme points of C^+ and C^-. Each consecutive pair defines a vertical slab containing at most two boundary segments: one from the boundary of C^+, and the other from the boundary of C^-. Notice that the boundaries of C^+ and C^- intersect in at most two points. Determination of whether two segments within a slab intersect can be performed in $O(1)$ sequential time. Hence, the intersection of C^+ and C^- can be done within

the complexity bounds of the convex-hull algorithm (see Exercise 6.12). Therefore, we have shown the following theorem.

Theorem 6.3: *Given a collection of n lines, each line defining a half-plane, it is possible to specify the intersections of all these half-planes in* $O(\log n)$ *time, using a total of* $O(n \log n)$ *operations.* □

PRAM Model: Our algorithm can be implemented on the CREW PRAM model, since the convex-hull algorithm requires only concurrent-read capability. The other operations required can be also implemented on the CREW PRAM model. □

6.2.3 TWO-VARIABLE LINEAR PROGRAMMING

The algorithm for computing the intersection of half-planes can be used to solve the two-variable linear-programming problem, which can be defined as follows:

Minimize $cx + dy$
Subject to $a_i x + b_i y + c_i \leq 0, 1 \leq i \leq n.$

The n inequalities $a_i x + b_i y + c_i \leq 0$, where $1 \leq i \leq n$, are called the **constraints,** and the function $cx + dy$ is called the **objective function.** The **feasible region** P is the set of points satisfying all the n constraints. Clearly, P is the intersection of a set of half-planes; hence, P is convex.

Solving a two-variable linear program amounts to determining a point $(x_0, y_0) \in P$ that minimizes the objective function. But the objective function can be viewed as a family F of *parallel* lines defined by $cx + dy = \lambda$, where λ is a parameter belonging to the field $\mathcal{R}$ of reals. Since P is convex, it is not difficult to verify that the objective function will be minimized at one of the extreme points of P.

EXAMPLE 6.6:

Consider the following linear program:

Minimize $-x + y$
Subject to:

$$\begin{aligned} x + y - 5 &\leq 0 \\ 3x + y - 9 &\leq 0 \\ y - 4 &\leq 0 \\ -x &\leq 0 \\ -y &\leq 0 \end{aligned}$$

The feasible region is the convex polygon shown in Fig. 6.9, which has five extreme points. The family of lines corresponding to the objective function is given by $\{L_\lambda : -x + y = \lambda\}$. Figure 6.9 shows the three lines L_5, L_4, L_{-3} corresponding to $\lambda = 5, 4, -3$, respectively. The optimal value of the objective function is equal to -3, and this value is achieved by the supporting line L_{-3} passing through the extreme point $(3, 0)$. Note that the other supporting line L_4 passes through the vertex $(0, 4)$ and achieves the maximum value of 4 for the objective function. □

In the two-variable case considered in this section, we can construct the feasible region explicitly by using the algorithm to determine the intersection of a set of half-planes. This algorithm takes $O(\log n)$ time, using a total of $O(n \log n)$ operations, where n is the number of constraints. Determining a solution to the linear program amounts to finding the minimum of $cx + dy$ for all (x, y) ranging over the extreme points of the feasible region. Since there are $O(n)$ such points, this task takes $O(\log n)$ time, using $O(n)$ operations. We therefore have the following theorem.

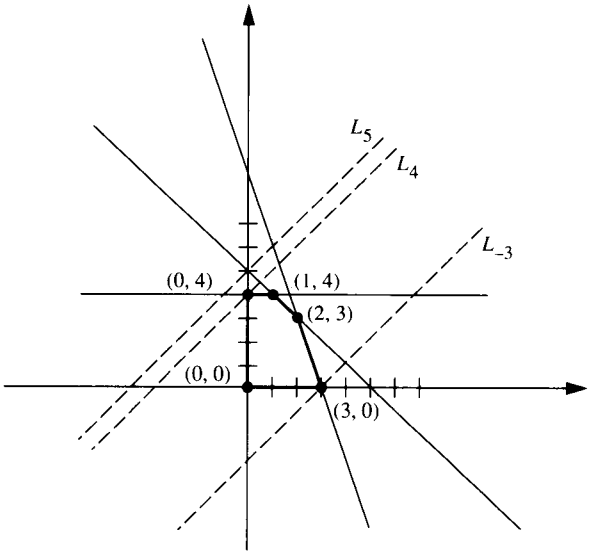


FIGURE 6.9
The boundary of the feasible region of the two-variable linear program of Example 6.6 (shown in dark lines). The three lines L_5, L_4, and L_{-3} correspond to the parameterized objective function $-x + y = \lambda$, for the values of $\lambda = 5$, 4, -3, respectively.

Theorem 6.4: *The two-variable linear program with n constraints can be solved on the CREW PRAM in $O(\log n)$ time, using a total of $O(n \log n)$ operations.* □

Note that this algorithm for the two-variable linear-programming problem is *not* optimal. In fact, linear-time sequential algorithms exist for the fixed-dimension linear-programming problem. An improvement over the given algorithm is suggested in Exercise 6.21.

Remark 6.1: The explicit construction of the feasible region for higher-dimensional linear programming is not computationally possible, since the number of extreme points can grow exponentially with the number of variables. The well-known **simplex method** solves a linear program by moving from one extreme point to another *adjacent* extreme point (ignoring degeneracy) until a local (also global, in this case) minimum is found, without explicitly constructing the feasible region. □

6.3 Plane Sweeping

Plane sweep is an important paradigm in computational geometry that is used to handle many problems in the plane. The main idea is fairly intuitive. Given a set of geometric objects in the plane, imagine a vertical line sweeping the plane from left to right. For each position of the sweep line, we can deduce information relevant to the desired solution by examining the intersections of the sweep line with the geometric objects. We need only to position the sweep-line at certain locations, called **critical points,** where possibly useful information can be extracted.

In this section, we consider an efficient parallel implementation of the plane-sweep method in the case when the basic geometric objects are horizontal and vertical line segments. We note that the basic technique can be extended to handle more general geometric objects, as will be pointed out later.

EXAMPLE 6.7:

Given a set of horizontal and vertical segments, consider the problem of determining, for each vertical segment, the number of horizontal segments intersecting it. We give a sketch of a sequential sweep-line method to solve this problem.

In this case, the critical points correspond to the endpoints of the segments. We start by sorting all the critical points according to their x coordinates. We then sweep the plane, starting from the endpoint with the smallest x coordinate. To store the relevant information generated during our plane sweep, we use a balanced binary tree that supports the operations of "find," "insert," and "delete" in logarithmic time. Each leaf of the binary tree corresponds to the y coordinate of a left endpoint of a horizontal segment that has been encountered but that has a right endpoint not yet processed. Each internal node contains the number of descendants in the subtree rooted at that node.

At each critical point, if the point is the left endpoint of a horizontal segment, we insert its y coordinate into the binary tree. If it is the right endpoint of a horizontal segment, then the corresponding y value is deleted from the tree. If we encounter a vertical segment, then we search the binary tree for the number of y values between the two y values of the segment's endpoints. Clearly, we can find the number of horizontal lines intersecting each vertical line in this fashion. It follows that the handling of each endpoint takes $O(\log n)$ sequential time, where n is the total number of segments. Hence, this sequential algorithm runs in $O(n \log n)$ time. □

As can be seen from Example 6.7, it is not obvious how the plane-sweep strategy can be implemented efficiently for parallel processing. We next describe the plane-sweep tree, which will allow us to process all the sweep lines at the critical points concurrently.

6.3.1 THE PLANE-SWEEP TREE

Let $S = \{s_1, s_2, \ldots, s_n\}$ be a set of nonintersecting horizontal segments. Assume that no two endpoints have the same x coordinate. We define the **plane-sweep tree** of S as follows.

Let $x_1 < x_2 < \ldots < x_{2n}$ be the x coordinates of all the endpoints. These $2n$ points on the x axis form $2n + 1$ intervals. Let T be a balanced binary tree constructed on these $2n + 1$ intervals. With each leaf u, associate the corresponding interval, denoted by $[a_u, b_u]$. An internal node v corresponds to the interval that is the union of the intervals associated with its descendants. We need to store additional information at each node v.

Let v be an arbitrary node of T whose interval is given by $[a_v, b_v]$. Let $\prod_v$ be the vertical strip $[a_v, b_v] \times (-\infty, +\infty)$. A segment s_i **covers** v if it spans $\prod_v$ but not $\prod_z$, where $z = p(v)$, the parent of v. Define $H(v)$ and $W(v)$ as follows:

$$H(v) = \{s_i \mid s_i \text{ covers } v\},$$

$$W(v) = \left\{s_i \mid s_i \text{ has at least one endpoint in } \prod_v\right\}.$$

Note that, if s_i has an endpoint on the boundary of $\prod_v$ and extends entirely outside $\prod_v$, then s_i will *not* be included in $W(v)$. We assume that the segments in each of $H(v)$ and $W(v)$ are *sorted by their ordinate values*. We call such a tree the *plane-sweep* tree of S.

EXAMPLE 6.8:

Consider the three segments s_1, s_2, and s_3 shown in Fig. 6.10(a). The x coordinates of the endpoints of s_i are denoted by x_{i1} and x_{i2} such that $x_{i1} < x_{i2}$. The corresponding sweep tree is shown in Fig. 6.10(b). The two lists $H(v)$ and $W(v)$ are given next to each node v. □

Lemma 6.5: *Let T be the plane-sweep tree of a set of n nonintersecting horizontal segments, and let $H(v)$ and $W(v)$ be the lists stored at each node v. Then, each of the sums $\sum_v |H(v)|$ and $\sum_v |W(v)|$ is of size $O(n \log n)$.*

Proof: Each segment can cover at most two nodes at each level, given that a segment that covers a node v cannot cover its parent. Hence, the sum of the sizes of the lists at each level is $O(n)$, and thus $\sum_v |H(v)| = O(n \log n)$.

Since each segment has two endpoints, no segment can appear more than twice in all the $W(v)$'s at any given level. Hence, $\sum_v |W(v)| = \sum_v |H(v)| = O(n \log n)$, and the lemma follows. □

We now show how to construct $H(v)$ and $W(v)$ efficiently for each node v in T.

We start by examining the lists $W(v)$. Suppose that v is a leaf corresponding to the interval $[x_i, x_{i+1}]$. In this case, $W(v)$ consists of at most two segments: one has an endpoint whose abscissa is equal to x_i, and the other has an endpoint whose abscissa is equal to x_{i+1}. Hence, computing $W(v)$ for each leaf v can be done in $O(1)$ time using a linear number of operations.

Consider an arbitrary internal node v whose left and right children are u and w, respectively. Any segment in $W(v)$ must be either in $W(u)$ or in $W(w)$. Conversely, a segment in $W(u)$ or in $W(w)$ must be in $W(v)$. Hence, $W(v)$ is the sorted list that we obtain by merging $W(u)$ and $W(w)$ according to the ordinate values of the segments. If we merge the lists bottom up, one level at a time, the resulting algorithm will run in more time than the $O(\log n)$ time for which we are aiming. However, our pipelined merge-sort algorithm (Algorithm 4.4) computes the sorted lists generated in all the nodes of an arbitrary binary tree. In our case, it leads to an algorithm that computes all the $W(v)$'s in $O(\log n)$ time, using a total of $O(n \log n)$ operations.

The computation of $H(v)$ will use the W arrays generated previously. Let v be an arbitrary node, and let z be the parent of v. It is clear that every segment in $H(v)$ must appear in $W(z)$. Our goal is to select the subset of edges in $W(z)$ that forms $H(v)$.

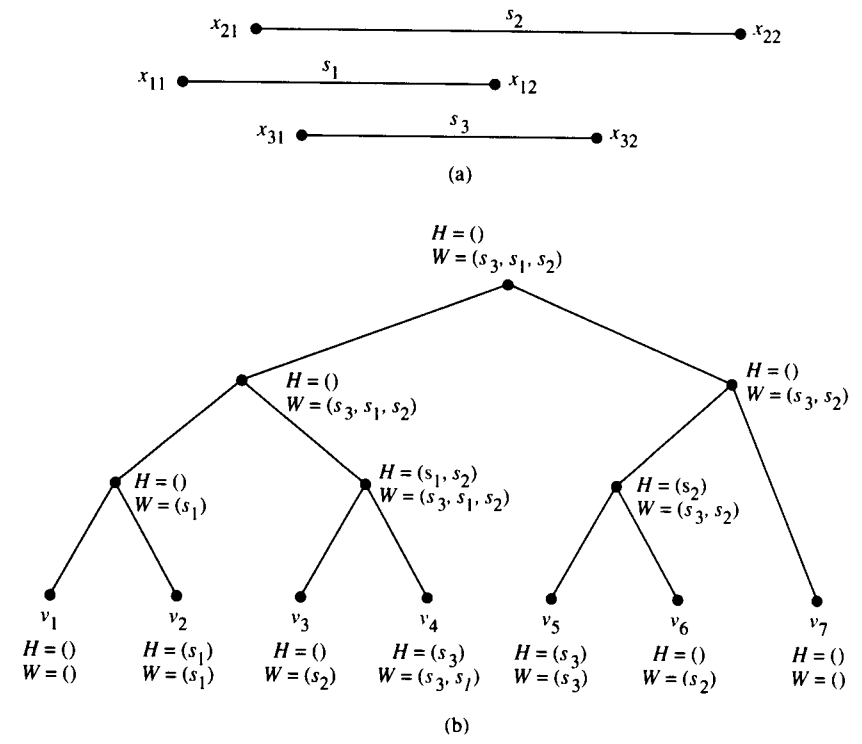


FIGURE 6.10
The sweep tree of the three segments for Example 6.8. (a) Three horizontal segments s_1, s_2, and s_3. (b) The plane-sweep tree constructed on the set of seven intervals induced by the six endpoints of s_1, s_2, and s_3 when the latter are sorted by their x values. Each node v represents an interval $[a_v, b_v]$ on the x axis and contains two lists H and W sorted by their y values. Consider, for example, the node $p(v_3)$, the parent of v_3; it corresponds to the interval $[x_{21}, x_{12}]$. Only s_1 and s_2 cover the corresponding vertical strip, and hence $H = (s_1, s_2)$. Each of the input segments has an endpoint in the vertical strip of $p(v_3)$, and hence $W = (s_3, s_1, s_2)$.

We can test whether a segment of $W(z)$ belongs to $H(v)$, in $O(1)$ sequential time, by comparing the x coordinates of its endpoints with $[a_v, b_v]$ and $[a_w, b_w]$, where w is the sibling of v. We can then mark each such segment that belongs to $H(v)$.

It follows that, for each node v, we can mark all the elements of $W(z)$ that belong to $H(v)$ in $O(1)$ time, using a linear number of operations, where $z = p(v)$. Since $\sum_z |W(z)| = O(n \log n)$, the total number of operations

required is $O(n \log n)$. Note that $H(r) = \emptyset$, where r is the root of T. Each internal node will then contain a marked copy for its left child, and a marked copy for its right child. In addition, the marked elements in each $W(z)$ appear in order sorted by their y values. Using the prefix-sums algorithm (Algorithm 2.1), we can then compact each marked copy in parallel in $O(\log n)$ time, using a total of $O(n \log n)$ operations, since $\sum_{v \in T} |W(v)| = O(n \log n)$ (see Exercise 6.23). Therefore, we have the following theorem.

Theorem 6.5: *Given a set S of n horizontal segments, we can construct the plane-sweep tree of S in $O(\log n)$ time, using a total of $O(n \log n)$ operations.* □

PRAM Model: The plane-sweep algorithm requires concurrent-read capability, since our version of the pipelined merge sort has this requirement. Moreover, marking the elements of $W(z)$ that belong to $H(v)$ in $O(1)$ time requires the concurrent access to the interval $[a_v, b_v]$. Hence, this algorithm runs on the CREW PRAM model within the stated asymptotic bounds. □



6.3.2 AN APPLICATION OF THE PLANE-SWEEP TREE

In this section, we consider an application of the plane-sweep tree. Other applications are covered in the Exercises 6.27, 6.28, 6.29, and 6.30.

Let $S = \{s_1, s_2, \ldots, s_n\}$ be a set of nonintersecting horizontal segments. We wish to determine, for each endpoint p of a segment in S, the **closest segment directly below** p, if such a segment exists. We show how to solve this problem in $O(\log n)$ time, using a total of $O(n \log n)$ operations, by using the plane-sweep tree. This algorithm can be used, for example, to generate the escape line segments of a set of rectilinear obstacles in VLSI routing, as defined in Exercise 6.30.

EXAMPLE 6.9:

Consider the horizontal segments shown in Fig. 6.11. For each endpoint p, a dotted line is drawn from p to the closest segment directly below p, whenever the latter exists. □

As before, we are assuming that no two endpoints have the same x coordinate. Let T be the plane-sweep tree associated with the set S. Recall that the vertical strip $\prod_v = [a_v, b_v] \times (-\infty, +\infty)$ is associated with node v, which also holds the two sets $W(v)$ and $H(v)$ sorted by their y coordinates.

We augment each node v with the array $rank(W(v) : H(v))$—that is, the array consisting of the ranks according to the y values of the segments of $W(v)$ in the sorted array $H(v)$. We can obtain his information easily while constructing

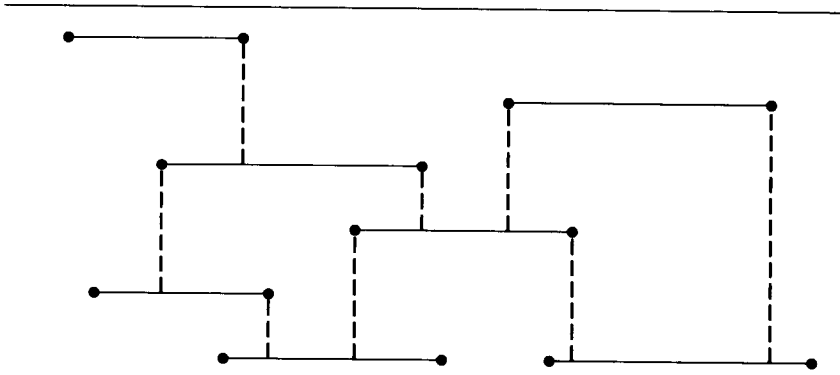


FIGURE 6.11
The closest segment directly below each endpoint. For each endpoint, a dashed line indicates the corresponding segment.

$H(v)$ (see Exercise 6.24). We can also generate it after T is constructed by using an optimal merging algorithm (say, the one discussed in Section 2.4) applied to all the nodes of T concurrently. Since $\sum_{v \in T} |W(v)|, \sum_{v \in T} |H(v)| = O(n \log n)$, we can do this task within the resource bounds required by the algorithm to build T.

We are now ready to describe the procedure to find for each endpoint p of a segment s_p the closest segment in S directly below p. We use the notation $y(s)$ to denote the y value of the horizontal segment s.

Lemma 6.6: *Let v be the leaf of the plane-sweep tree T such that $\prod_v$ contains p and s_p spans $\prod_v$, where p is an endpoint of s_p. Let P be the tree path from v to the root. Then a segment s is directly below p if and only if s appears on a list $H(u)$, for some node u on P, and $y(s) < y(s_p)$.*

Proof: Let s be an arbitrary segment that is directly below p. Then, s has to span $\prod_v$, since there are no endpoints in the interior of $\prod_v$ and no two endpoints have the same x coordinate. It follows that, for some node u on P, s spans $\prod_u$ but s does not span $\prod_{p(u)}$, where $p(u)$ is the parent of u. Hence, s appears on the list $H(u)$. Clearly, we must have $y(s) < y(s_p)$.

Conversely, suppose that s appears on the list $H(u)$, for some node u on the path P, such that $y(s) < y(s_p)$. Then, the segment s has to span $\prod_u$ which must contain $\prod_v$. Using the two facts that s spans $\prod_v$ and that $y(s) < y(s_p)$, we conclude that s must be directly below p. □

Lemma 6.6 indicates that we can solve the problem of determining, for an endpoint p, the closest segment directly below it by searching the lists $H(u)$

for the segment with the largest y value less than $y(s_p)$, for all u on the path P. Searching each such list can be done in $O(\log n)$ sequential time with the binary search method. Since there are $O(\log n)$ such lists along the path P, this approach leads to an $O(\log^2 n)$ sequential-time algorithm to identify the desired segment for each endpoint. The resulting algorithm to handle all the endpoints concurrently runs in $O(\log^2 n)$ time, using $O(n \log^2 n)$ operations. We next improve this algorithm.

Since we have already generated the array $rank(W(v) : H(v))$, for each node v, we can, in $O(1)$ sequential time, locate the predecessor $pred_u(s_p)$ of s_p in $H(u)$, for each u on P (recall that s_p is in $W(u)$ for each u on P). The predecessor of s_p in $H(u)$ is precisely the segment s' in $H(u)$ with the largest y value satisfying $y(s') < y(s_p)$. We then obtain the solution by computing the minimum of the y values of the segments $\{pred_u(s_p)\}_{u \in P}$. This step clearly takes $O(\log n)$ sequential time for each endpoint. Therefore, we have the following theorem.

Theorem 6.6: *Given a set of n nonintersecting horizontal segments, the problem of determining, for each endpoint p, the closest segment directly below p can be solved in $O(\log n)$ time, using a total of $O(n \log n)$ operations.* □

PRAM Model: Construction of the plane-sweep tree requires the CREW PRAM model, as we saw earlier. The additional processing clearly can be handled on this model. Therefore, our algorithm runs on the CREW PRAM model within the stated asymptotic bounds. □

The computation involved in determining the segment in $H(v)$ directly below a given point p, for all v such that $p \in \prod_v$, is called the **multilocation** of p in T. This type of computation arises in important problems, some of which are mentioned in the Exercises 6.29 and 6.30.

Note that the plane-sweep tree for an arbitrary set of segments (not necessarily horizontal) can be constructed in $O(\log n)$ time, using a total of $O(n \log n)$ operations. Moreover, the multilocation of a point p can also be done in $O(\log n)$ sequential time when the sweep tree is properly augmented. The details required for these constructions are involved, although the techniques used are essentially the same as those presented here.

6.4 Visibility Problems

Given a set F of *forbidden* curves and a point p in the plane, a point q is **visible** from p if the line segment pq does not intersect any curve in F. If F consists of line segments, the points visible from p form a (possibly unbounded) polygon, called the **visibility polygon** of p.

In this section, we consider the case when F is a set of *nonintersecting* line segments, and show how to determine the visibility polygon of a point p in $O(\log n)$ time, using a total of $O(n \log n)$ operations. To achieve this solution, we will make critical use of the general pipelined divide-and-conquer strategy introduced in Chapter 4. A review of this strategy is in order.

6.4.1 A QUICK REVIEW OF PIPELINED MERGE SORT

We presented in Section 4.3.2 the pipelined divide-and-conquer strategy as applied to the following problem. Given a binary tree T such that each leaf u contains a list $L[u]$ drawn from a linearly ordered set, we wish to determine, for each internal node v, the sorted list $L[v]$ that contains all the elements stored in the subtree rooted at v. The pipelined merge-sort algorithm handles such a computation in $O(h(T) + \log m)$ iterations, each iteration requiring $O(1)$ time, where $h(T)$ is the height of T and m is the maximum number of elements in any of the initial lists. During each iteration, the list $L[v]$ stored at node v is updated by a merging operation involving samples of the lists stored in the children nodes. The important point for us here is that, when the algorithm terminates, we can assume that we have the sorted list $L[v]$ at each node v and the rank information $rank(L[u], L[v])$, where u is a child of v. In addition, the lists generated at each sibling pair of nodes are cross-ranked (see Remark 4.6). Therefore, if u and w are the children of v, then the *predecessor* of an element p of $L[u]$ in the list $L[w]$ can be determined in $O(1)$ sequential time. The predecessor of an element p of $L[u]$ in $L[w]$ is the largest element of $L[w]$ that is smaller than p.

6.4.2 THE LOWER ENVELOPE OF A SET OF SEGMENTS

Let $S = \{s_1, s_2, \ldots, s_n\}$ be a set of nonintersecting segments, except possibly at their endpoints. Let p be the point at negative infinity below all the segments. Essentially, the same algorithm can be used to handle the case when p is an arbitrary finite point; the details are left to Exercise 6.32. The visibility polygon can now be characterized by the **lower envelope** of the n segments consisting of the pieces of segments from S that are visible from $-\infty$. The lower envelope can be specified by a list $(p_0 = -\infty, p_1, p_2, \ldots, p_s)$, where the p_i's are endpoints of segments with increasing x coordinates, together with a corresponding array *vis* of visible segments such that, for each p_i, $vis(p_i)$ indicates the line segment visible in the interval $(x(p_i), x(p_{i+1}))$, where $0 \leq i < s$, and $x(p_i)$ is the x coordinate of p_i. We note that $x(-\infty) = -\infty < x(p_i)$ for any p_i, and $vis(p_s) = +\infty$.

EXAMPLE 6.10:

Consider the set of segments $S = \{s_1, s_2, s_3, s_4, s_5\}$ shown in Fig. 6.12(a). The lower envelope is defined by the list $(-\infty, p_1, p_2, p_3, p_5, p_6, p_8, p_9, p_{10})$, and by the array of visible segments given by $vis = (+\infty, s_1, +\infty, s_2, s_3, s_4, +\infty, s_5, +\infty)$. For example, in the interval $(x(p_3), x(p_5))$, the segment s_2 is visible, and hence $vis(p_3) = s_2$. On the other hand, there is no segment visible in the interval $(x(p_8), x(p_9))$, and hence $vis(p_8) = +\infty$. □

To compute the lower envelope, we will use a divide-and-conquer strategy that is similar to the one used in the merge-sort algorithm (Algorithm 4.4). We start by computing the trivial lower envelope of each segment separately. We then compute the lower envelopes of pairs of segments, followed by computing the lower envelopes of pairs of pairs of segments. We continue in this fashion until the lower envelope of all the segments has been generated. We will describe our algorithm as the process of constructing a

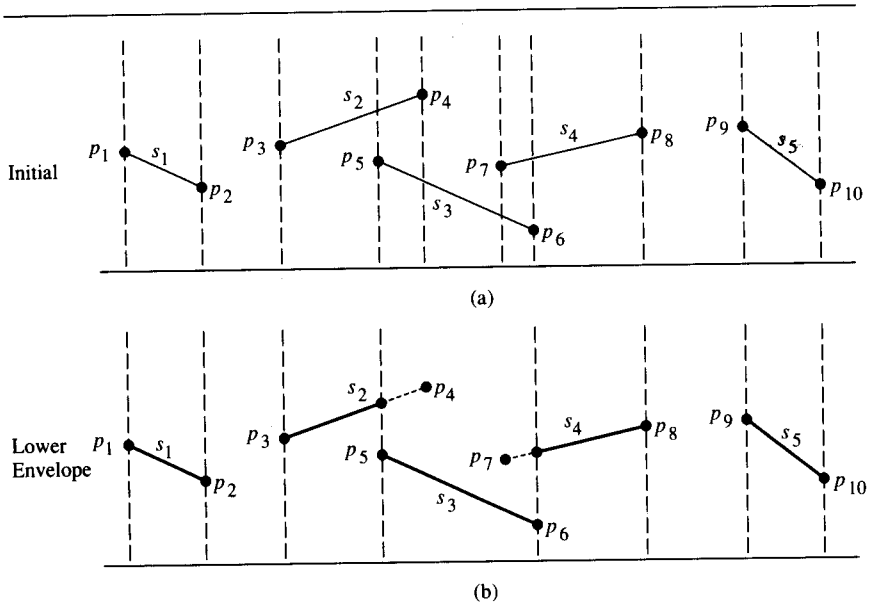


FIGURE 6.12
The lower envelope of a set of segments. (a) The initial set of segments. (b) The corresponding lower envelope specified by the list of endpoints $L = (-\infty, p_1, p_2, p_3, p_5, p_6, p_8, p_9, p_{10})$ and the label list $vis = (+\infty, s_1, +\infty, s_2, s_3, s_4, +\infty, s_5, +\infty)$. For example, the segment visible in the interval (p_5, p_6) is s_3.

balanced binary tree whose leaves correspond to the initial segments such that, when the algorithm terminates, each internal node will contain the lower envelope of the segments in its subtree. The details are given next.

Let $S = \{s_1, s_2, \ldots, s_n\}$ be the given set of segments such that no two segments intersect, except possibly at their endpoints. For clarity, assume that the x coordinates of the endpoints are distinct. Our aim is to construct the following binary tree based on the elements of S.

Let T be a balanced binary tree with n leaves. Leaf v_i contains the list $L[v_i] = (-\infty, p_{i1}, p_{i2})$, where p_{i1} and p_{i2} are the two endpoints of s_i such that $x(p_{i1}) < x(p_{i2})$, and where the corresponding lower envelope is given by $(+\infty, s_i, +\infty)$. We denote this fact by the labels: $vis(-\infty, v_i) = +\infty$, $vis(p_{i1}, v_i) = s_i$, and $vis(p_{i2}, v_i) = +\infty$. Note that $L[v_i]$ and the corresponding *vis* array provide the correct lower envelope of the segment s_i.

Let v be an internal node. Then, $L[v]$ consists of $p_0 = -\infty$ followed by all the endpoints stored in the subtree rooted at v, ordered by increasing x coordinates. Also, for each such p_i, we have $vis(p_i, v) = s_{j_i}$, indicating that segment s_{j_i} is visible (with respect to the segments stored in the subtree rooted at v) in the interval $(x(p_i), x(q_i))$, where q_i is the endpoint that comes after p_i in $L[v]$. If p_i is the last element in $L[v]$, then $vis(p_i, v) = +\infty$. Hence, the root r will hold the list consisting of $p_0 = -\infty$ followed by all the $2n$ endpoints sorted by their x coordinates, and the corresponding *vis* array.

We now discuss how to construct the binary tree T. We can obtain the list $L[v]$ at each node v by merging the sublists at the children nodes according to their x coordinates. Hence, the pipelined merge-sort algorithm (Algorithm 4.4) can be used to compute the lists $L[v]$, for all nodes v, in $O(\log n)$ time, using a total of $O(n \log n)$ operations. The only detail left is to show how to generate efficiently the labels defined by the array *vis*.

Let v be an arbitrary internal node whose children are the nodes u and w. After the pipelined merge-sort algorithm (Algorithm 4.4) terminates, we have the sorted lists $L[v]$, $L[u]$, and $L[w]$ such that $L[v] = L[u] \cup L[w]$. In addition, the two lists $L[u]$ and $L[w]$ are cross-ranked, and are each ranked in $L[v]$. Suppose that we are given the two arrays $vis(., u)$ and $vis(., w)$. We show that $vis(., v)$ can be generated in $O(1)$ time, using $O(|L[v]|)$ operations.

Let p be an endpoint in $L[v]$ that comes from $L[u]$, and let p' be the element of $L[u]$ that follows p. Then $vis(p, u)$ is the visible segment in the interval $(x(p), x(p'))$ among the segments stored in the subtree rooted at u. Since $L[u]$ and $L[w]$ are cross-ranked, we can find, in $O(1)$ sequential time, the predecessor of p in $L[w]$, which will be denoted by $pred_w(p) = q$. Let q' be the successor of q on the list $L[w]$. We clearly have $x(q) < x(p) < x(q')$. Moreover, $vis(q, w)$ is the visible segment over the interval $(x(q), x(q'))$ among the segments stored in the subtree rooted at w.

At node v, the successor of p in $L[v]$ must be either p' or q', whichever has the smaller x value. Hence, $vis(p, v)$ must the visible segment in the

interval $I_p = (x(p), \min\{x(p'), x(q')\})$ among all the segments in the subtree rooted at v. Over the interval I_p, there are two possible candidates—namely, $vis(p, u)$ or $vis(q, w)$. Therefore, $vis(p, v)$ is the lower segment at $x(p)$ among $vis(p, u)$ and $vis(q, w)$.

Similarly, given a point p of $L[v]$ that comes from $L[w]$, the visible segment $vis(p, v)$ is the lower segment at $x(p)$ of $vis(p, w)$ and $vis(pred_u(p), u)$.

Therefore, we can compute, in $O(1)$ sequential time, for an arbitrary point p of $L[v]$, the value $vis(p, v)$. It follows that the list vis can be generated at each node v from the lists $vis(., u)$ and $vis(., w)$ in $O(1)$ time, using $O(|L[v]|)$ operations.

In summary, all the labels of the lists vis can be generated bottom up, level by level, in $O(\log n)$ time, using a total of $O(n \log n)$ operations.

There is one additional detail that we must address. For the root r of the tree T, the length of $L[r]$ and $vis(., r)$ is $2n + 1$ corresponding to the $2n$ endpoints of the n input segments and to the leftmost endpoint $-\infty$. We can compact each of these two lists in $O(\log n)$ time, using a total of $O(n)$ operations, by using the prefix-sums algorithm (Algorithm 2.1). This step yields the lower envelope of the set of input segments.

EXAMPLE 6.11:

Consider the set $S = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8\}$ of segments shown in Fig. 6.13(a). The endpoints of segment s_i are denoted by p_{i1} and p_{i2} such that $x(p_{i1}) < x(p_{i2})$. Initially, each segment s_i is stored at a leaf v_i as follows: $L[v_i] = (-\infty, p_{i1}, p_{i2})$, and $vis(., v_i) = (+\infty, s_i, +\infty)$.

Let u be the parent of v_1 and v_2. We construct the list $L[u]$ by merging $L[v_1]$ and $L[v_2]$ according to the x coordinates, and hence $L[u] = (-\infty, p_{11}, p_{21}, p_{12}, p_{22})$. The array $vis(., u)$ can be determined as follows. First, we set $vis(-\infty, u) = +\infty$ (which is always true for any node u.) Now p_{11} comes from v_1, and its predecessor in v_2 is $-\infty$. Hence, $vis(p_{11}, u)$ is the lower of $vis(p_{11}, v_1) = s_1$ and $vis(-\infty, v_2) = +\infty$; that is, $vis(p_{11}, u) = s_1$. Since the predecessor of p_{21} in $L[v_1]$ is p_{11}, we have that $vis(p_{21}, u)$ is the lower of $vis(p_{21}, v_2) = s_2$ and $vis(p_{11}, v_1) = s_1$, which is s_2 since at p_{21}, $y(s_2) < y(s_1)$. Hence, $vis(p_{21}, u) = s_2$, and so on. We thus obtain $vis(., u) = (+\infty, s_1, s_2, s_2, +\infty)$. Notice that s_2 appears consecutively twice: the first time corresponds to the endpoint p_{21}, and the second time corresponds to the endpoint p_{12}.

We now illustrate the last merging step. Let the children of the root r be the nodes a and b. Then, $L[a] = (-\infty, p_{11}, p_{21}, p_{12}, p_{22}, p_{31}, p_{32}, p_{41}, p_{42})$, $vis(., a) = (+\infty, s_1, s_2, s_2, +\infty, s_3, +\infty, s_4, +\infty)$, $L[b] = (-\infty, p_{51}, p_{52}, p_{61}, p_{62}, p_{71}, p_{72}, p_{81}, p_{82})$, and $vis(., b) = (+\infty, s_5, +\infty, s_6, +\infty, s_7, +\infty, s_8, +\infty)$. Now, we can check that $L[r] = (-\infty, p_{11}, p_{51}, p_{21}, p_{12}, p_{52}, p_{61}, p_{22}, p_{31}, p_{62}, p_{71}, p_{32}, p_{72}, p_{81}, p_{41}, p_{82}, p_{42})$, and $vis(., r) = (+\infty, s_1, s_1, s_2, s_2, s_2, s_2, s_6, s_3, s_3, s_3, s_7, +\infty, s_8, s_4, s_4, +\infty)$. Let us verify the value of $vis(p_{22}, r)$.

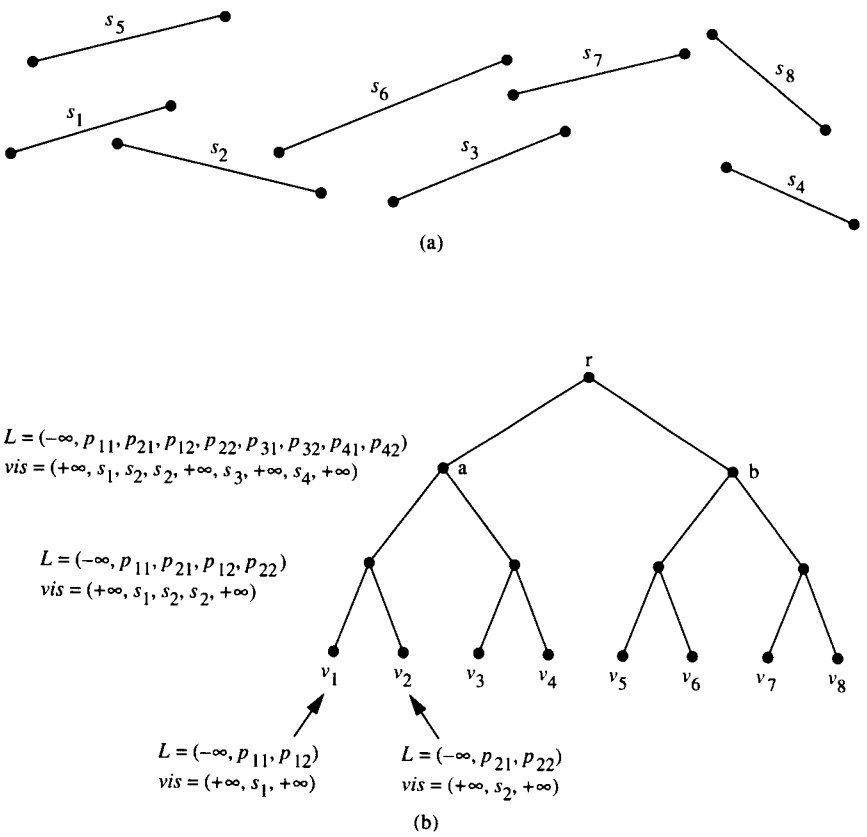


FIGURE 6.13
Lower envelope of segments for Example 6.11. (a) A set of segments. (b) Part of the tree in the lower-envelope computation. Each leaf v_i represents the lower envelope of segment s_i, and each internal node v represents the lower envelope of the segments stored in the subtree rooted at v. For example, the lists shown at node a represent the lower envelope of the segments s_1, s_2, s_3, and s_4.

The predecessor of p_{22} in $L[b]$ is p_{61}. Now $vis(p_{22}, a) = +\infty$, and $vis(p_{61}, b) = s_6$, and hence $vis(p_{22}, r) = s_6$. Compacting the lists generated at the root r, we obtain the list of endpoints $(-\infty, p_{11}, p_{21}, p_{22}, p_{31}, p_{32}, p_{72}, p_{81}, p_{41}, p_{42})$, and the vis array $(+\infty, s_1, s_2, s_6, s_3, s_7, +\infty, s_8, s_4, +\infty)$. □

We therefore have the following theorem.

Theorem 6.7: *Given a set of n nonintersecting line segments, the pipelined merge-sort algorithm (Algorithm 4.4) can be used to determine the lower envelope of these segments in $O(\log n)$ time, using a total of $O(n \log n)$ operations.* □

PRAM Model: The lower envelope algorithm can be implemented on the CREW PRAM model because our procedure to update the labels requires only concurrent-read capability, as several elements of a list may have the same predecessor in another list. □

6.5 Dominance Counting

Given two points p and q in the plane, p **dominates** q (or q is **dominated** by p) if $x(p) \geq x(q)$ and $y(p) \geq y(q)$. The fact that a point p dominates another point q is denoted by $p > q$. For a given set V of points, the pair $(V, >)$ is a partial order. That is, the relation $>$ is reflexive, antisymmetric, and transitive. This concept turns out to be useful, as the following two examples illustrate.

EXAMPLE 6.12:

Let R be a rectangle whose sides are parallel to the coordinate axes, and let $Q = \{q_1, q_2, \ldots, q_l\}$ be a set of points in the plane. We wish to find the number n of points of Q contained in R.

Let the vertices of R be (p_1, p_2, p_3, p_4) ordered by traversing the boundary of R counterclockwise, starting from the upper-right corner (as in Fig. 6.14.) Let $d(p_i) = |\{q_j \in Q \mid p_i > q_j\}|$; that is, $d(p_i)$ is the number of points from Q dominated by p_i. These points are contained in the quadrant whose upper-right corner is p_i and whose sides are parallel to the coordinate axes. Then, it is simple to check that $n = d(p_1) + d(p_3) - d(p_2) - d(p_4)$.□

EXAMPLE 6.13:

Given a set Q of points in the plane, a point p of Q will be called a **maximal element** if there exists no $q \in Q$ such that $q \neq p$ and $q > p$.

Consider the quadrant of Q formed by the two points of maximum x and y coordinates, respectively as shown in Fig. 6.15. In this quadrant, the set of maxima (marked with crosses) forms an "approximation" to the corresponding upper hull. Note that every point of the upper hull is a maximal element in this quadrant. Similar comments hold for other quadrants, given an appropriate assignment of + and − signs to each of the coordinates of the points in Q. □

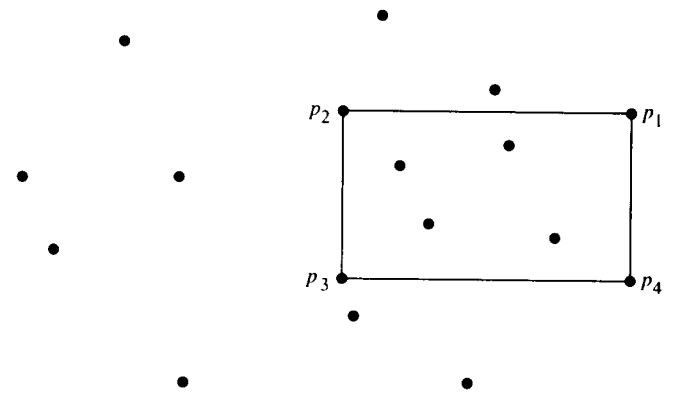


FIGURE 6.14
Determination of the number of points inside a rectangle. The number of points contained in the rectangle is given by $d(p_1) + d(p_3) - d(p_2) - d(p_4)$, where $d(p_i)$ is the number of points dominated by p_i.

6.5.1 THE DOMINANCE-COUNTING PROBLEM

In the remainder of this section, we consider the following **dominance-counting problem.** Given two sets $A = \{p_1, p_2, \ldots, p_m\}$ and $B = \{q_1, q_2, \ldots, q_l\}$ of points in the plane, determine, for each q_i in B, the number $d(q_i)$ of points in A that are dominated by q_i. We assume, for simplicity, that no two points of A and B have the same x or y coordinate.

As in the previous section, we shall use the cascading divide-and-conquer approach. The pipelined merge-sort algorithm (Algorithm 4.4) will be executed on a balanced binary tree built on $A \cup B$. At the same time, a set of labels will be generated such that the labels at the root will provide the solution to the dominance-counting problem.

Let $R = \{r_1, r_2, \ldots, r_n\}$ be the points in $A \cup B$ such that $y(r_1) < y(r_2) < \cdots < y(r_n)$, where $n = l + m$. The initial preprocessing required to sort all the points according to their y values can be performed in $O(\log n)$ time, using a total of $O(n \log n)$ operations. Let T be a balanced binary tree with n leaves $v_1, v_2, \ldots, v_n$ such that leaf v_i represents point r_i, where $1 \leq i \leq n$. Each leaf v_i contains a list $U[v_i] = (-\infty, r_i)$.

Each internal node v of T will contain the list $U[v]$ consisting of $-\infty$ followed by all points stored in the subtree rooted at v *sorted* by their x coordinates. These lists can be generated by using the pipelined merge-sort algorithm (Algorithm 4.4) in $O(\log n)$ time, using $O(n \log n)$ operations.

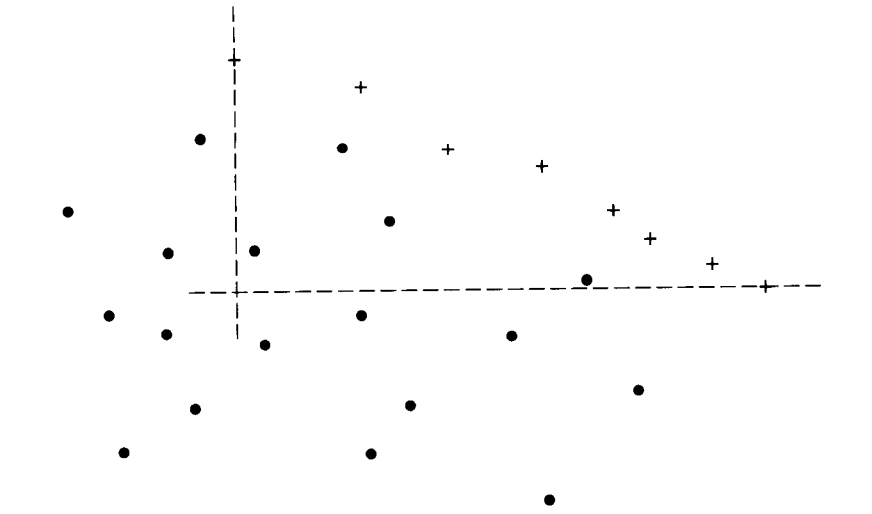


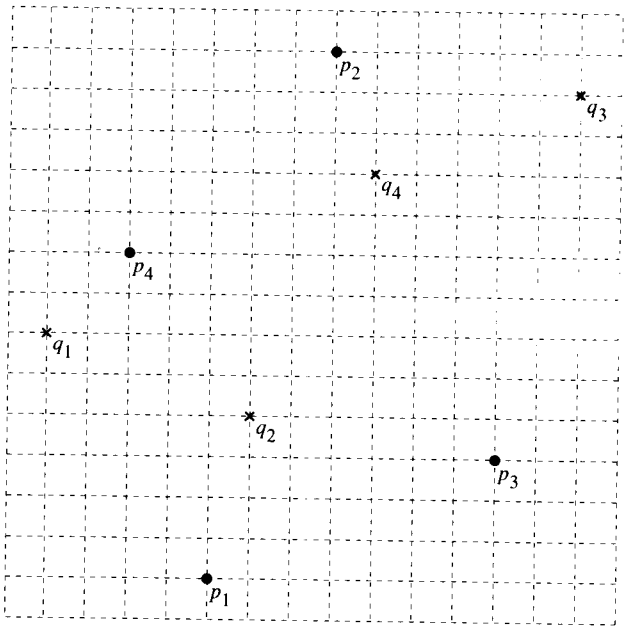
FIGURE 6.15
The quadrant for Example 6.13. The maxima of a set of points residing in the indicated quadrant are marked with crosses.

We now attach two labels to each element in the list $U[v]$ generated at node v. For each point r_i of $U[v]$, we let $d'(r_i, v)$ be the number of elements of A *preceding* r_i in $U[v]$. That is, $d'(r_i, v) = |\{r \in U[v] \cap A \mid x(r) < x(r_i)\}|$. In addition, we let $d(r_i, v)$ be the number of points in $U[v] \cap A$ that are *dominated* by r_i. We initialize the labels at each leaf v_i as follows: $d'(r_i, v_i) = d'(-\infty, v_i) = d(r_i, v_i) = d(-\infty, v_i) = 0$.

Therefore, each node v of the binary tree T contains a list $U[v]$ of all the points in its subtree sorted according to their x values, and two integer lists $d'(., v)$ and $d(., v)$ of the same length.

EXAMPLE 6.14:

Consider the two sets of points $A = \{p_1, p_2, p_3, p_4\}$ and $B = \{q_1, q_2, q_3, q_4\}$ shown in Fig. 6.16(a). The list R, sorted by the y values, is given by $R = \{p_1, p_3, q_2, q_1, p_4, q_4, q_3, p_2\}$; hence, $U[v_1] = (-\infty, p_1)$, $U[v_2] = (-\infty, p_3)$, $U[v_3] = (-\infty, q_2)$, and so on. The complete tree is shown in Fig. 6.16(b), together with the lists $U[v]$, $d'(., v)$, and $d(., v)$ for each node v. The U and d lists generated at the root provide a complete answer to the dominance-counting problem relative to the sets A and B. □



(a)

FIGURE 6.16
Two sets of points for Example 6.14. (a) The two input sets $A = \{p_1, p_2, p_3, p_4\}$ (black dots) and $B = \{q_1, q_2, q_3, q_4\}$ (crosses) for the dominance-counting problem.

Let v be an internal node of T whose left and right children are u and w, respectively. Suppose that the labels $d'(., u)$, $d(., u)$, $d'(., w)$ and $d(., w)$ have already been computed. We show how to generate the arrays $d'(., v)$ and $d(., v)$ in $O(1)$ time, using $O(|U[v]|)$ operations.

The leftmost point $-\infty$ of each list $U[v]$ has the following labels: $d'(-\infty, v) = d(-\infty, v) = 0$, for every node v. Otherwise, the following lemma relates the value of a label at the node v to the values computed at the children nodes u and w.

Lemma 6.7: *Let T be the balanced binary tree built on the set $R = \{r_1, r_2, \ldots, r_n\}$. For an arbitrary internal node v, let u and w be its left and right children,*

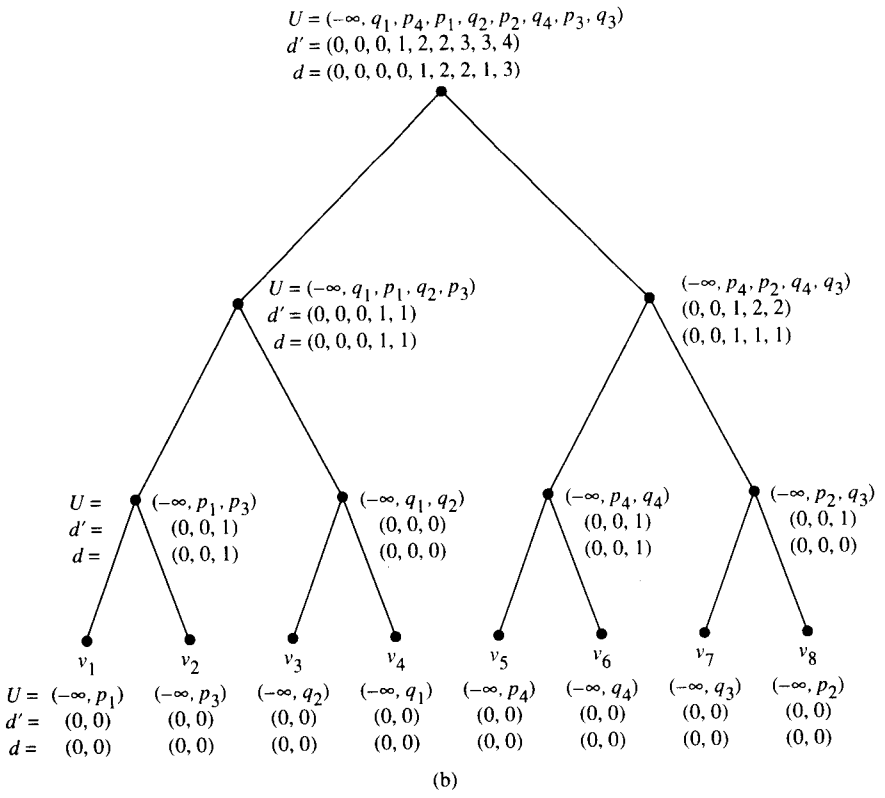


FIGURE 6.16 (*continued*)
(b) The binary tree corresponding to the inputs in (a). The leaves v_i correspond to all the points sorted according to their y values. The U list in each node v consists of all the points in the subtree rooted at v that are sorted according to their x values. The value $d'(r_i, v)$ of point r_i at node v is equal to the number of elements in $U[v] \cap A$ that precede r_i, and the value $d(r_i, v)$ is equal to the number of points in $U[v] \cap A$ that are dominated by r_i.

respectively. Then, the labels of a point r_i of the list $U[v]$ satisfy the following relations:

$$d'(r_i, v) = \begin{cases} d'(r_i, u) + d'(pred_w(r_i), w) + \chi_A(pred_w(r_i)) & \text{if } r_i \in U[u], \\ d'(r_i, w) + d'(pred_u(r_i), u) + \chi_A(pred_u(r_i)) & \text{if } r_i \in U[w]; \end{cases}$$

$$d(r_i, v) = \begin{cases} d(r_i, u) & \text{if } r_i \in U[u], \\ d(r_i, w) + d'(pred_u(r_i), u) + \chi_A(pred_u(r_i)) & \text{if } r_i \in U[w]; \end{cases}$$

where $\chi_A(r) = 1$ *if* $r \in A$ *and* $\chi_A(r) = 0$ *otherwise.*

Proof: We start by establishing the relation for the label $d'(r_i, v)$. Consider the case where $r_i \in U[u]$. Since $U[v]$ results from merging $U[u]$ and $U[w]$, $d'(r_i, v)$ is equal to the number of $r \in (U[u] \cup U[w]) \cap A$ such that $x(r) < x(r_i)$. The number of $r \in U[u] \cap A$ with $x(r) < x(r_i)$ is precisely $d'(r_i, u)$. On the other hand, the number of $r \in U[w] \cap A$ with $x(r) < x(r_i)$ is precisely $d'(pred_w(r_i), w) + \chi_A(pred_w(r_i))$. Therefore, $d'(r_i, v) = d'(r_i, u) + d'(pred_w(r_i), w) + \chi_A(pred_w(r_i))$. The case where $r_i \in U[w]$ can be treated in a similar fashion.

Consider now the label $d(r_i, v)$. If r_i comes from $U[u]$, then $d(r_i, v)$ must be equal to $d(r_i, u)$, since no element from $U[w]$ can be dominated by r_i, as each element of $U[w]$ has a y value larger than those of all the elements of $U[u]$. On the other hand, if r_i comes from $U[w]$, then all the points in $U[u]$ that have smaller x values are dominated by r_i, since r_i has already a larger y value. Therefore, the relations corresponding to $d(r_i, v)$ follow. □

Lemma 6.7 implies that the labels d' and d can be updated at each level in $O(1)$ time, using $O(n)$ operations. Since the root will contain the desired information, we have the following theorem.

Theorem 6.8: *Given a set* $A = \{p_1, p_2, \ldots, p_m\}$ *and a set* $B = \{q_1, q_2, \ldots, q_l\}$ *of points in the plane, the problem of determining, for each* $1 \le i \le l$, *the number* $d(q_i)$ *of points in A that are dominated by* q_i *can be solved in* $O(\log n)$ *time, using a total of* $O(n \log n)$ *operations, where* $n = l + m$. □

PRAM Model: Our algorithm consists of an application of the pipelined merge-sort algorithm (Algorithm 4.4) to generate the lists $U[v]$, followed by computation of the label lists d' and d bottom up, level by level. The computation of the d' and d labels as stated in Lemma 6.7 requires a concurrent-read capability, since several points may have the same predecessor. Therefore, our algorithm runs on the CREW PRAM model, within the stated resources. □

6.5.2 APPLICATIONS

We now consider several problems that can be handled efficiently using the parallel algorithm to solve the dominance-counting problem.

The first problem is stated in the following theorem.

Theorem 6.9: *Given a set A of m points in the plane, and a set* $\mathcal{R}$ *of l rectangles whose sides are parallel to the coordinate axes, we can determine, for each*

rectangle R in $\mathcal{R}$, the number of points of A that lie inside R, in $O(\log n)$ time, using a total of $O(n \log n)$ operations, where $n = m + l$.

Proof: Let B be the set of points that are the vertices of all the rectangles in $\mathcal{R}$. We solve the dominance-counting problem with respect to the pair A and B; that is, for each point $q \in B$, we determine the number $d(q)$ of points in A dominated by q.

Let R be the rectangle (q_1, q_2, q_3, q_4) whose vertices are ordered counterclockwise starting from the upper-right corner. As we saw in Example 6.12, the number of points of A contained in R is given by $d(q_1) + d(q_3) - d(q_2) - d(q_4)$. Hence, such a number can be computed in $O(1)$ sequential time for each rectangle, once all the d values have been determined. □

EXAMPLE 6.15:

For $1 \leq i \leq 4$, consider the set of rectangles $R_i = (q_{i1}, q_{i2}, q_{i3}, q_{i4})$ shown in Fig. 6.17. The points in A are shown as black dots. Let us determine the

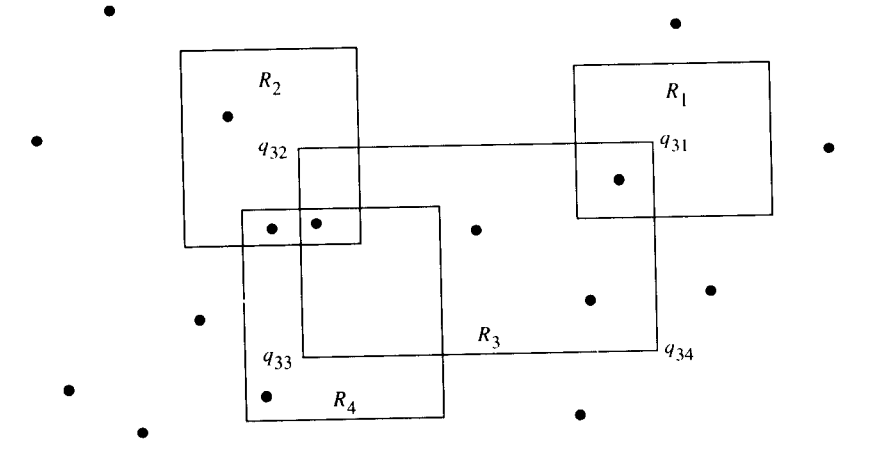


FIGURE 6.17
Determination of the number of points within each rectangle. The vertices of the rectangle R_3 are given by $(q_{31}, q_{32}, q_{33}, q_{34})$. The number of marked points dominated by a corner q_{3j} is denoted by $d(q_{3j})$. For the rectangle R_3, the number of points lying inside it is given by $d(q_{31}) + d(q_{33}) - d(q_{32}) - d(q_{34}) = 10 + 3 - 4 - 5 = 4$. Hence, once the d values are generated, the number of points lying inside each rectangle can be determined in $O(1)$ time, using a linear number of operations.

number of points of A in R_3. Since $d(q_{31}) = 10, d(q_{33}) = 3, d(q_{32}) = 4$, and $d(q_{34}) = 5$, we obtain $d(q_{31}) + d(q_{33}) - d(q_{32}) - d(q_{34}) = 4$, which is indeed the number of points of A contained in R_3. We use a similar computation for the other rectangles. □

The other problem that can be solved by using the parallel algorithm for dominance counting is stated in the next theorem.

Theorem 6.10: *Given a set S of n horizontal and vertical segments in the plane, the problem of determining, for each segment s, the number of segments in S that intersect s can be computed in $O(\log n)$ time, using a total of $O(n \log n)$ operations.*

Proof: Let U_L and U_R be, respectively, the sets of the left and the right endpoints of all the horizontal segments. Given an arbitrary point p, let $d_L(p)$ and $d_R(p)$ be, respectively, the numbers of points in U_L and U_R dominated by p. That is, $d_L(p)$ is the number of left endpoints of horizontal segments that are dominated by p, and $d_R(p)$ is the number of right endpoints of horizontal segments that are dominated by p. Then, $d_L(p) - d_R(p)$ is the number of horizontal segments below p that cross the vertical line through p.

Let V be the set of the endpoints of the vertical segments. We start by solving the dominance-counting problem with respect to the pair U_L and V, and to the pair U_R and V. Hence, for each endpoint $p \in V$, we have the values of $d_L(p)$ and $d_R(p)$. Consider now an arbitrary vertical segment $s = pq$, where $y(p) > y(q)$. Based on the observation made in the previous paragraph, the value $d_L(p) - d_R(p)$ is equal to the number of horizontal segments crossing the vertical line below p, and similarly, $d_L(q) - d_R(q)$ corresponds to the horizontal segments crossing the vertical line below q. Therefore, the number of horizontal segments crossing s is given by $(d_L(p) - d_R(p)) - (d_L(q) - d_R(q))$.

Hence, we can find, for each vertical segment, the number of horizontal segments that cross it, in $O(1)$ sequential time, once all the values of d_L and d_R have been computed. It follows that the problem of determining, for each vertical segment, the number of horizontal segments that cross it can be solved in $O(\log n)$ time, using $O(n \log n)$ operations.

In a similar fashion, we can find, for each horizontal segment, the number of vertical segments that cross it; therefore, the proof of the theorem follows. □

EXAMPLE 6.16:

Figure 6.18 shows two vertical segments pq and $p'q'$ and six horizontal segments $s_i = r_{i1}r_{i2}$, where $1 \leq i \leq 6$. Clearly, $U_L = \{r_{i1} \mid 1 \leq i \leq 6\}$, $U_R =$

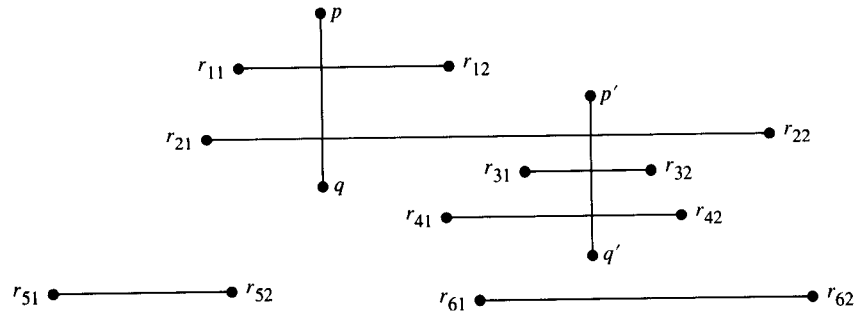


FIGURE 6.18
Intersections of a set of horizontal and vertical segments. Let $d_L(p)$ and $d_R(p)$ be, respectively, the numbers of the left and the right endpoints of horizontal segments dominated by p. Then, the number of horizontal segments crossing the vertical segment pq is given by $(d_L(p) - d_R(p)) - (d_L(q) - d_R(q))$.

$\{r_{i2} \mid 1 \leq i \leq 6\}$, and $V = \{p, q, p', q'\}$. The solution of the dominance-counting problem relative to U_L and V gives $d_L(p) = 3, d_L(q) = 1, d_L(p') = 5$, and $d_L(q') = 2$, and the solution to the other dominance-counting problem produces $d_R(p) = 1, d_R(q) = 1, d_R(p') = 1$, and $d_R(q') = 1$. Now $(d_L(p) - d_R(p)) - (d_L(q) - d_R(q)) = 2$, and precisely two horizontal segments cross pq. Similarly, $(d_L(p') - d_R(p')) - (d_L(q') - d_R(q')) = 3$. □

6.6 Summary

In this chapter, we have touched on a few fundamental problems in planar computational geometry. The divide-and-conquer strategy has been shown to yield efficient parallel algorithms for all the problems considered. The effective pipelining of the merging operations created by the divide-and-conquer strategy was critical in achieving the $O(\log n)$ time performance in all the algorithms except the convex-hull algorithm. The pipelined divide-and-conquer used in conjunction with the plane-sweep tree or augmented with labeling functions provides a powerful parallel technique for computational geometry. All the parallel algorithms described in this chapter run on the CREW PRAM in $O(\log n)$ time, using $O(n \log n)$ operations.

Table 6.1 provides a summary of the algorithms described in this chapter.

TABLE 6.1
ALGORITHMS INTRODUCED IN THIS CHAPTER.

Problem	Section	Time	Work	PRAM Model
Upper Common Tangent	6.1.3	$O(1)$	$O(n)$	CREW
Convex Hull	6.1.4	$O(\log n)$	$O(n \log n)$	CREW
Intersections of Half-Planes	6.2.1 and 6.2.2	$O(\log n)$	$O(n \log n)$	CREW
Two-Variable Linear Programming	6.2.3	$O(\log n)$	$O(n \log n)$	CREW
Construction of Plane-Sweep Tree	6.3.1	$O(\log n)$	$O(n \log n)$	CREW
Lower Envelope	6.4.2	$O(\log n)$	$O(n \log n)$	CREW
Dominance Counting	6.5.1	$O(\log n)$	$O(n \log n)$	CREW
Number of Points Inside a Rectangle	6.5.2	$O(\log n)$	$O(n \log n)$	CREW
Number of Horizontal Segments Intersecting Each Vertical Segment	6.5.2	$O(\log n)$	$O(n \log n)$	CREW

Exercises

6.1. Show how to modify appropriately the algorithm for finding the convex hull of a set of points (Algorithm 6.1), presented in Section 6.1, in the case where some points may have the same x or y coordinate.

6.2. Show that Algorithm 6.1 can be modified to run in $O\left(\frac{\log s \log t}{\log^2 p}\right)$ time, where p is the number of processors available. What are the corresponding running time and total number of operations when $p = \sqrt{n}$? Does your answer indicate that we can specify the upper hull of $S_1 \cup S_2$ using $O(n)$ operations? Explain your response.

6.3. Given a set S of n points in the plane, develop a parallel algorithm to compute the area of the convex hull of S. Your algorithm should run in $O(\log n)$ time, using a total of $O(n \log n)$ operations.

6.4. Given a simple polygon P (not necessarily convex) with n vertices and a point p in the plane, develop an $O(\log n)$ time algorithm to determine whether or not p is internal to P. Your algorithm should use a linear number of operations. *Hint:* Draw a horizontal line through p.

6.5. Let P be a convex polygon with n vertices given in a clockwise cyclic ordering, and let Q be a set of m points external to P. You are to determine the convex hull of the set of points $P \cup Q$.

a. Prove that, if $m = 1$, then the convex hull can be determined in $O(\log n)$ sequential time.

b. Develop an $O(\log m)$ time parallel algorithm for the general case. What is the total number of operations used? Assume that the convex hull of Q does not intersect P.

6.6. Given a convex polygon P with n vertices listed in a clockwise cyclic ordering, develop an algorithm to find the *diameter* of P—that is, the maximum distance between any two of P's vertices. Your algorithm should run in $O(\log n)$ time on the CREW PRAM, using a linear number of operations. *Hint:* A pair of vertices that have parallel supporting (tangent) lines is called *antipodal*. You need to consider only the $O(n)$ antipodal pairs. Consider the list A of edge angles and the list B when the angles are rotated 180 degrees. Neighbors in the merged list correspond to antipodal pairs.

6.7. Given a set S of n points in the plane, consider the problem of determining a pair of points of S that are farthest apart.

a. Show that the diameter of the convex hull of S is equal to the maximum distance between any two points of S.

b. Develop an $O(\log n)$ time algorithm to identify a pair of points in S, the members of which are farthest apart. What is the total number of operations used? What is the PRAM model you need? *Hint:* Use your answer to Exercise 6.6.

6.8. Let $L_i : \{x \mid a_i x + b_i \leq 0\}$ be a set of half-lines, where $1 \leq i \leq n$.

a. Show how to determine their intersections in $O(\log \log n)$ time, using a linear number of operations. Which PRAM model do you need?

b. Show how to derive an optimal fast solution for the one-variable linear-programming problem.

6.9. Let $\{I_j \mid 1 \leq j \leq n\}$ be a set of intervals on the line. Develop an $O(\log n)$ time algorithm to determine whether or not any two intervals overlap. Use a total of $O(n \log n)$ operations. Specify the PRAM model you need.

6.10. Consider a set of half-planes $H(L_i)$, where $1 \leq i \leq n$, and let T be the transformation that maps a point $p = (a, b)$ into the line $T(p)$ defined by $y = ax + b$, and a line $L : y = ax + b$ into the point $T(L) = (-a, b)$, as in Section 6.2. Does the set $T(\cap H(L_i))$ constitute the convex hull of the points $\{T(L_i)\}$? Is the set $T(\cap H(L_i))$ necessarily convex? Explain your answers.

6.11. Show that the problem of determining the polygonal chain defining the boundary of the intersections of a set of half-planes requires $\Omega(n \log n)$ operations. *Hint:* Suppose you wish to sort the n numbers $q_1, q_2, \ldots, q_n$. Define the n lines $L_i : y = 2q_i x - q_i^2$, where $1 \leq i \leq n$. Consider $\cap_{1 \leq i \leq n} H^+(L_i)$.

6.12. Given two convex polygons P and Q whose vertices are listed in a clockwise cyclic ordering, develop an $O(\log n)$ time algorithm to specify the convex region formed by the intersection of P and Q. Your algorithm must use a linear number of operations. *Hint:* Draw a vertical line through each vertex, and consider each slab formed by two successive vertical lines.

6.13. Solve Exercise 6.5 for the case where Q is an arbitrary set of m points. Use the algorithm developed in Exercise 6.12.

6.14. Let $P = (p_1, p_2, \ldots, p_n)$ and $Q = (q_1, q_2, \ldots, q_n)$ be two convex polygons, and let $d_b(P, Q)$ be the minimum distance between a boundary point of P and a boundary point of Q. Show that computing $d_b(P, Q)$ is at least as difficult as computing the OR function of n variables. *Hint:* Let P be a regular convex n-gon and let $y = x_1 + x_2 + \cdots + x_n$ be the OR function. With each x_i, associate a vertex q_i of Q such that q_i is slightly above the middle point of the ith segment of P if $x_i = 1$; otherwise, q_i is slightly below the middle point.

6.15. Let P be a convex polygon whose n vertices are given in a counterclockwise cyclic ordering, and let q be a point external to P. Develop an algorithm to determine the point p in P such that the Euclidean distance between p and q is minimum. Your algorithm should run in $O\left(\frac{\log n}{\log p}\right)$ time on the CREW PRAM, using $1 < p < n$ processors.

6.16. Given a set S of n points and a point $p \notin S$ in the plane, define the **Voronoi polygon** $V(p)$ associated with p to be the locus of points in the plane that are closer to p than they are to any point in S. See Fig. 6.19 for an example.

a. Show that $V(p)$ is the intersection of n half-planes.

b. Deduce an $O(\log n)$ time algorithm to identify the boundary of $V(p)$.

6.17. Let $P = (v_0, \ldots, v_{n-1})$ be a convex polygon whose vertices are given in a counterclockwise cyclic ordering. The *all-nearest-neighbors* problem is to find, for each vertex v_i, a vertex $v_{j(i)}, j(i) \neq i$, such that the Euclidean distance between v_i and $v_{j(i)}$ is minimal. The polygon P has the *semicircle* property if (1) the two farthest vertices of P are the endpoints—say, v_i and v_{i+1}—of an edge of P, and (2) all the vertices of P lie inside a circle with diameter $d(v_i, v_{i+1})$, the distance between v_i and v_{i+1}.

a. Show that, if P has the semicircle property, then $j(i) = i - 1$ or $i + 1$ for any vertex v_i of P.

b. Show how to decompose P into four convex subpolygons, each with the semicircle property.

c. *Develop an $O(\log \log n)$ time algorithm to solve the all-nearest-neighbors problem for an arbitrary convex polygon P. Your algorithm should use a linear number of operations. *Hint:* Use the optimal $O(\log \log n)$ time merging algorithm found in Section 4.2.3.

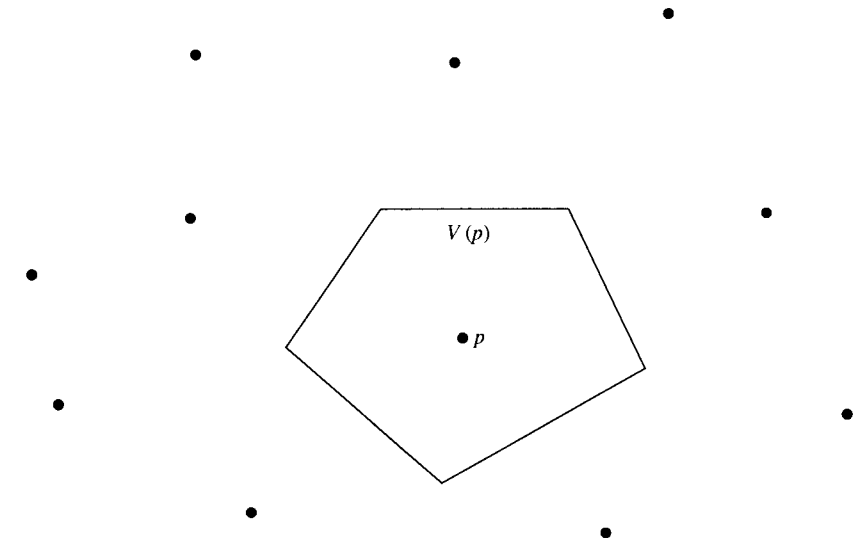


FIGURE 6.19
The Voronoi polygon $V(p)$ associated with the point p. The polygon consists of all the points in the plane that are closer to p than they are to any of the marked points.

6.18. Let $S = \{p_1, p_2, \ldots, p_n\}$ be a set of points such that $x(p_1) < x(p_2) < \cdots < x(p_n)$, where n is a power of 2. You wish to find two points p_i and p_j such that the Euclidean distance $d(p_i, p_j)$ is minimum. Let $S_1 = \{p_1, \ldots, p_{\frac{n}{2}}\}$, and $S_2 = \{p_{\frac{n}{2}+1}, \ldots, p_n\}$, and let δ_1(respectively δ_2) be the minimum distance between any pair of points in S_1(respectively S_2). Set $\delta = \min\{\delta_1, \delta_2\}$.

- **a.** Show that the number of points in S_2 that are within distance δ of a point $p_i \in S_1$ is at most 6.
- **b.** Develop a parallel algorithm to determine the closest pairs, based on a divide-and-conquer strategy. What is the running time of your algorithm? Your algorithm should use $O(n \log n)$ operations.

6.19. Let S_1 and S_2 be two sets of points in the plane. Consider the problem of finding a line L separating S_1 and S_2 whenever possible; for example, all the points of S_1 appear below L, and all the points of S_2 above L. Show how to solve this problem in $O(\log n)$ time, using a total of $O(n \log n)$ operations, where $n = |S_1| + |S_2|$. *Hint:* You may want to reformulate this problem as a linear program.

6.20. Let $f(x) = \max_{1 \le i \le n} a_i x + b_i$, for a given set of coefficients $\{a_i, b_i\}$, where $1 \le i \le n$. Note that $f(x)$ is continuous, is convex, and is made up of pieces that are line segments. Consider the problem of computing a point x_0 on the real line such that $f(x_0)$ is minimum.

a. Develop an $O(\log n)$ time algorithm to solve this problem. The total number of operations should be $O(n \log n)$.

b. Can you develop a polylogarithmic-time algorithm that uses $O(n)$ operations? Explain your answer. *Hint:* Pair up lines and find their intersections. Identify the median of the intersection abscissae, and remove a constant fraction of the lines.

6.21. Use the algorithm of Exercise 6.20(b) to develop a polylogarithmic-time algorithm for solving the two-variable linear-programming problem, using $O(n)$ operations.

6.22. Let $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$ be a set of continuous real functions defined on an interval $I \subseteq \mathcal{R}$, the field of real numbers. Assume that, for $i \neq j$, the functions f_i and f_j intersect in at most a constant number s of points on I (you may want to view each f_i as a polynomial of degree $\le s$). The *lower envelope* of $\mathcal{F}$ is the function $g(x) = \min_i\{f_i(x)\}$, for $x \in I$. The function $g(x)$ defines a continuous curve in the plane made up of pieces $f_{i_1}, f_{i_2}, \ldots, f_{i_m}$, where each $f_{i_j} \in \mathcal{F}$ achieves the minimum over some interval I_j. If the intervals are sorted from left to right, the sequence of functions forms an (n, s) **Davenport–Schinzel sequence.**

Let $m(n, s)$ denote the maximum number of pieces. Then, it is known that $m(n, s) < c(s)n \log^* n$. Moreover, $m\left(\frac{n}{2}, s\right) < \frac{1}{2} m(n, s)$. Using a divide-and-conquer strategy, develop an algorithm to compute the lower envelope of $\mathcal{F}$. Assume that each f_i can be described in $O(1)$ space and can be computed in $O(1)$ sequential time at any given point, and that the output is an array E of size m, where $E(k) = (i_k, x_k)$, implying that the envelope consists of the piece f_{i_k} over the interval $[x_k, x_{k+1}]$. Your algorithm should run in $O(\log^2 n)$ time, using $O(n \log n)$ operations.

6.23. Suppose that you are given an array A of length n with some of its elements marked, and you want to compute the prefix sums of each subarray consisting of the elements of A between two consecutive marked elements (the marked elements themselves are assigned arbitrarily to the subarrays). Show how to perform this task in $O(\log n)$ time, using a total of $O(n)$ operations.

6.24. Show how to construct the plane-sweep tree T of a set of n nonintersecting horizontal segments such that the array $rank(W(v) : H(v))$ will be available at each node v. Your algorithm should run in $O(\log n)$ time, using $O(n \log n)$ operations. Do not use the optimal merging algorithm found in Section 4.2.3.

6.25. Let $S = \{s_1, \ldots, s_n\}$ be a set of nonintersecting horizontal segments. Consider the problem of constructing the plane-sweep tree T of S with the following additional list $B(v)$ stored at each node v in sorted order of the y values:

$$B(v) = \left\{ s_i \mid s_i \text{ has both endpoints in } \prod_v \right\}$$

Moreover, if the children of v are u and w, then the arrays $rank(H(v) : B(u))$, $rank(H(v) : B(w))$, $rank(B(u) : H(v))$, and $rank(B(w) : H(v))$ should also be computed. Show how to construct T with the additional information in $O(\log n)$ time, using a total of $O(n \log n)$ operations.

6.26. Let S be a set of nonintersecting segments in the plane such that no two endpoints have the same x coordinate. Consider the problem of constructing the plane-sweep tree T as introduced in Section 6.3. Note that, in this case, the segments are not necessarily horizontal. For each node v whose left and right children are u and w, respectively, define $I(v)$ to be the set of segments with one endpoint in $\prod_u$ and the other in $\prod_w$ (sorted by the vertical boundary separating $\prod_u$ and $\prod_w$).

a. Develop an algorithm to construct $I(v)$ for each node v starting from the initial segments. Your algorithm should run in $O(\log n)$ time, using $O(n \log n)$ operations.

b. *Develop an $O(\log n \log \log n)$ time algorithm to construct T (and $H(v)$, for each v). What is the total number of operations used?

6.27. *Let T be the plane-sweep tree of a set of nonintersecting segments. We would like to augment each node v of T with a list $A(v)$ (in addition to $H(v)$) so that the multilocation of a point p can be done in $O(\log n)$ sequential time. Show how to accomplish this goal. What are the resources required by your algorithm? *Hint:* Try to propagate samples of $H(v)$ into the nodes of the subtree rooted at v.

6.28. Assume that you have the plane-sweep tree T of a set of nonintersecting segments augmented as indicated in Exercise 6.27. Show how to compute, for each endpoint p, the closest segment directly above p. Your algorithm should run in $O(\log n)$ time, using a total of $O(n \log n)$ operations.

6.29. A graph $G = (V, E)$ is *planar* if it can be embedded in the plane without crossings. A planar graph can always be embedded in the plane such that each embedded edge is a line segment. Such a planar embedding determines what is called a *planar subdivision*. Figure 6.20 illustrates a planar subdivision.

Given such a planar subdivision S with n edges, consider the problem of building a data structure such that the problem of determining the face of S containing an arbitrary point p can be solved in $O(\log n)$ sequential

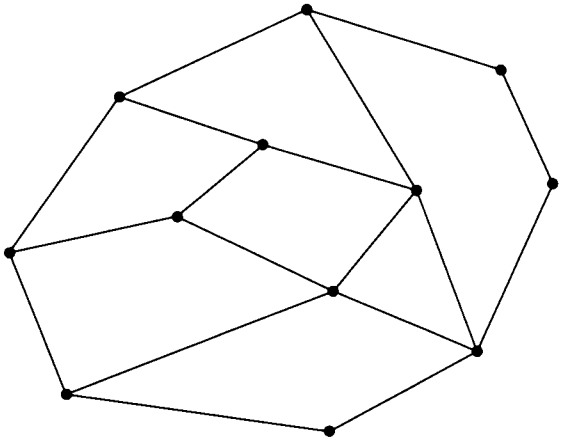


FIGURE 6.20
A planar subdivision.

time. Show how to use the augmented plane-sweep tree introduced in Exercise 6.27 to accomplish this task.

6.30. Let G be a grid of size $m \times m$, and let R be a set of nonintersecting rectangles whose sides lie on gridline segments. For each side s of a rectangle in R, you wish to find the maximal line segment $L(s)$ in G parallel to s and at a unit distance of s such that $L(s)$ does not intersect any rectangle in R. Show how to determine all the segments L in $O(\log n)$ time, using a total of $O(n \log n)$ operations, where n is the total number of rectangles in R. *Hint:* Use the plane-sweep tree. Handle carefully the sides that are within 1 unit distance of each other.

6.31. Let P be a simple polygon with n vertices. The **kernel** of P, denoted by $K(P)$, is the set of points q in P such that, for any point z on the boundary of P, the segment zq lies entirely in P. Figure 6.21 shows a polygon with its kernel marked by shading.

a. Show that $K(P)$ is the intersection of n half-planes defined by the edges of P.

b. Deduce an $O(\log n)$ time algorithm to compute $K(P)$. What is the total number of operations used?

6.32. Let $S = \{s_1, s_2, \ldots, s_n\}$ be a set of nonintersecting segments, and let p be an arbitrary point in the plane. Modify the algorithm described in Section 6.4 to determine the visibility polygon of p. Your algorithm should run in $O(\log n)$ time, using a total of $O(n \log n)$ operations. *Hint:* Start by sorting the endpoints radially around p.

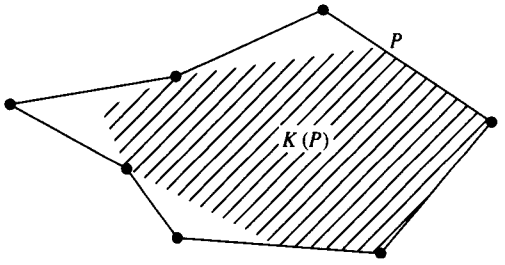


FIGURE 6.21
A polygon and its kernel (shaded region).

6.33. Let S be a set of n horizontal and vertical segments in the plane. Use the plane-sweep tree to determine, for each vertical segment, the number of horizontal segments intersecting it. What is the running time of your algorithm? Compare the performance of your algorithm with the algorithm referred to in Theorem 6.10.

6.34. Given a set of n points in the plane, develop an $O(\log n)$ time algorithm to identify all the maximal points with respect to the dominance relation $>$ introduced in Section 6.5. What is the total number of operations used? What is the PRAM model used?

6.35. Let $S = \{p_1, p_2, \ldots, p_n\}$ be a set of points in $\mathcal{R}^3$, where $\mathcal{R}$ is the field of real numbers. Assume that no two points have the same x, or y, or z coordinates. A point p_i dominates p_j if $x(p_i) > x(p_j), y(p_i) > y(p_j)$, and $z(p_i) > z(p_j)$. A point p_i is a *maximum* if it is not dominated by any other point in S. Develop an $O(\log n)$ time algorithm to find the maxima of S. Your algorithm should use $O(n \log n)$ operations. *Hint:* Use a pipelined divide-and-conquer strategy.

Bibliographic Notes

Some of the early work on parallel algorithms for computational geometry appeared in Chow's dissertation [9]. The refined divide-and-conquer algorithm for computing the convex hull presented in Section 6.1 is taken from [6]. Other optimal $O(\log n)$ time parallel algorithms also appeared in [1, 5]. If the points are given in sorted order according to their x values, the convex hull can be determined in $O(\log n)$ time, using a total of $O(n)$ operations [15]. The problem of determining the intersection of a set of half-planes is well known to be equivalent to the convex-hull problem (for more details, see, for example [14, 18]). Our presentation follows Atallah and Goodrich [7]. Linear-time sequential algorithms for linear programming with a fixed number of variables appeared in [13, 16, 17]. An optimal $O(\log n)$ time parallel algorithm for the

two-variable case appeared in [12]. Plane sweeping is a well-known powerful technique in sequential computational geometry, and has been used to solve numerous geometric problems. Bentley introduced the idea of a segment tree, the precursor of the plane-sweep tree. The latter data structure was first introduced by Aggarwal et al. [1], and was later improved by Atallah et al. [4]. The plane-sweep tree has been used to solve many problems, including the trapezoidal decomposition problem (Exercise 6.28), the planar point-location problem (Exercise 6.29), and the problem of detecting whether any two segments of a given set of segments intersect. The pipelined divide-and-conquer technique augmented with labeling functions was also introduced by Atallah et al. [4], and was used to solve the visibility problem of Section 6.4 and the dominance-counting problem of Section 6.5.

In this chapter, we have touched on only a few techniques for computational geometry. Other parallel techniques not mentioned in the text include geometric hierarchies [11], searching of monotone arrays [2, 3], and randomization [19, 20]. Geometric hierarchies and randomization techniques will be discussed in Chapter 9. For additional reading about sequential techniques, the books [14, 18] are recommended.

A solution to the problem of computing the diameter of a convex polygon (Exercise 6.6) can be found in [1]. The fact that the problem of computing the minimum distance between two convex polygons is at least as difficult as computing the OR function was observed in [6]. Substantial work has been done on the computation of Voronoi diagrams (Exercise 6.16), some of which is described in the books [14, 18]. The solution sketched in Exercise 6.17 to solve the all-nearest-neighbors problem for a convex polygon is taken from [21]. The parallel complexity of the problem of finding the closest pair of points (Exercise 6.18) has been examined in [5, 10]. The solution to Exercise 6.25 can be found in [8]. An optimal $O(\log n)$ time algorithm to compute the kernel of a polygon described in Exercise 6.31 appeared in [10]. A solution to the three-dimensional maxima problem given in Exercise 6.35 can be found in [4].

References

1. Aggarwal, A., B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap. Parallel computational geometry. *Algorithmica*, 3(3):293–327, 1988.
2. Aggarwal, A., M. M. Klawe, S. Moran, P. Shor, and R. Wilber. Geometric applications of a matrix searching algorithm. *Algorithmica*, 2(2):209–233, 1987.
3. Aggarwal, A., and J. Park. Parallel searching in multidimensional monotone arrays. *Journal of Algorithms*, in press, 1991.
4. Atallah, M., R. Cole, and M. Goodrich. Cascading divide-and-conquer: a technique for designing parallel algorithms. *SIAM J. Computing*, 18(3):499–532, 1989.
5. Atallah, M. J., and M. T. Goodrich. Efficient parallel solutions to some geometric problems. *Journal of Parallel and Distributed Computing*, 3(4):492–507, 1986.
6. Atallah, M. J., and M. T. Goodrich. Parallel algorithms for some functions of two convex polygons. *Algorithmica*, 3(4):535–548, 1988.
7. Atallah, M. J., and M. T. Goodrich. *Deterministic parallel computational geometry*, in Synthesis of Parallel Algorithms, J. Reif, ed. Morgan Kaufmann, San Mateo, CA, 1991.
8. Chandran, S. *Merging in parallel computational geometry.* PhD thesis, Department of Computer Science, University of Maryland, College Park, MD, 1989.

9. Chow, A. *Parallel algorithms for geometric problems.* PhD thesis, University of Illinois, Urbana-Champaign, IL, 1980.
10. Cole, R., and M. T. Goodrich. Optimal parallel algorithms for point-set and polygon problems. *Algorithmica*, in press, 1991.
11. Dadoun, N., and D. Kirkpatrick. Parallel processing for efficient subdivision search. In *Proceedings Third ACM Symposium on Computational Geometry*, Waterloo, Ontario, Canada, 1987, pp. 205–214.
12. Deng, X. An optimal parallel algorithm for linear programming in the plane. *Information Processing Letters*, 35(4):213–217, 1990.
13. Dyer, M. E. Linear time algorithms for two and three-variable linear programs. *SIAM J. Computing*, 13(1):31–45, 1984.
14. Edelsbrunner, H. *Algorithms in Combinatorial Geometry.* Springer-Verlag, New York, 1987.
15. Goodrich, M. T. Finding the convex hull of a sorted point set in parallel. *Information Processing Letters*, 26(4):173–179, 1987.
16. Megiddo, N. Linear time algorithm for liner programming in $\mathcal{R}^3$ and related problems. *SIAM J. Computing*, 12(4):759–776, 1983.
17. Megiddo, N. Linear programming in linear time when the dimension is fixed. *JACM*, 31(1):114–127, 1984.
18. Preparata, F. P., and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, 1985.
19. Reif, J. H., and S. Sen. Polling: a new random sampling technique for computational geometry. In *Proceedings Twenty-First Annual ACM Symposium on Theory of Computing*, Seattle, WA, 1989, pp. 394–404.
20. Reif, J. H., and S. Sen. Optimal randomized parallel algorithms for computational geometry. *Algorithmica*, in press, 1991.
21. Schieber, B., and U. Vishkin. Finding all nearest neighbors for convex polygons in parallel: A new lower bound technique and a matching algorithm. *Discrete Applied Mathematics*, 29(1):97–111, 1990.

7

Strings

Many important computational problems can be formulated as a search for the occurrences of a given set of objects, called **patterns,** in a given subject. Specific application areas where such problems arise include text editing, computer vision, speech recognition, and molecular biology. In this chapter, we deal with strings and their exact occurrences in other strings. Powerful algorithmic tools, based on mathematical properties of strings, are well known and can be used to handle efficiently many processing tasks on strings.

Most of the algorithmic techniques for exact **string matching** seem to revolve around the *periodic properties* of strings. Such properties are introduced first, followed by the development of an optimal logarithmic-time parallel string-matching algorithm. The main paradigm used consists of *pattern analysis*, followed by *text analysis*. The output generated at the end of the pattern-analysis phase is a table containing information related to the periodic structure of the pattern. The text-analysis phase uses this table to identify all the occurrences of the pattern in the text. We also introduce a data structure, called the *suffix tree*, that can be used to support many string manipulations. An efficient parallel algorithm to set up such a data structure is presented, as are several of its applications.

7.1 Preliminary Facts About Strings

Let Σ be an **alphabet** consisting of a finite number of *symbols*. A **string** Y over Σ is a finite sequence of elements from Σ. The **length** of Y, denoted by $|Y|$, is the total number of elements in the sequence defining Y. If X and Y are strings, the **concatenation** of X and Y is the string XY; that is, it is the sequence defining X followed by the sequence defining Y.

In describing our algorithms on strings, we assume that a string Y is represented by an array such that $Y(i)$ is the ith element of Y, where $1 \leq i \leq |Y|$.

Let Y be a string of length m. For any pair of indices i and j such that $1 \leq i \leq j \leq m$, the subarray $Y(i:j)$ defines a **substring** of Y. Hence, a substring of Y is any contiguous block of characters in Y. A substring defined by $Y(1:i)$, for some i such that $1 \leq i \leq m$, is called a **prefix** of Y, whereas a substring of the form $Y(j:m)$, for some j such that $1 \leq j \leq m$, is called a **suffix** of Y.

A string X **occurs** in Y at position i if X is equal to the substring of Y of length $|X|$ starting at location i. More formally, $X(j) = Y(i + j - 1)$, for all indices j such that $1 \leq j \leq |X|$. We also say that X **matches** Y at position i.

EXAMPLE 7.1:

Let $\Sigma = \{a, b, c\}$, and let Y be the string $Y = aabcabccaa$. Then, $X = abc$ is a substring of Y that occurs at positions 2 and 5. The prefix $Y(1:5)$ is the substring $aabca$, and the suffix $Y(5:10)$ is the substring $abccaa$. □

7.1.1 PERIODICITIES IN STRINGS

The periodic properties of strings provide the basis for certain algorithmic tools for the efficient manipulation of strings. In this section, we introduce the notion of the period of a string, and we study a few related properties.

Let Y be a string of length m. A substring X of a string Y will be called a **period** of Y if $Y = X^k X'$, where X^k is the string consisting of X concatenated with itself k times, k being a positive integer, and X' is a prefix of X. Hence, Y consists of several consecutive copies of X, followed by a prefix of X. It is easy to check that X is a period of Y if and only if Y is a prefix of XY.

The definition of a period has the following important implication. Suppose a copy of Y is placed above Y but is shifted i positions from the left end of Y, for some i such that $1 \leq i \leq m - 1$ (see Fig. 7.1). Then, if the overlapping portions are identical, Y has a period consisting of the prefix $Y(1:i)$. In other words, if, for some i, the two substrings $Y(i + 1:m)$ and $Y(1:m - i)$ are equal, then $Y(1:i)$ is a period of Y. Note that Y is always a period of itself.

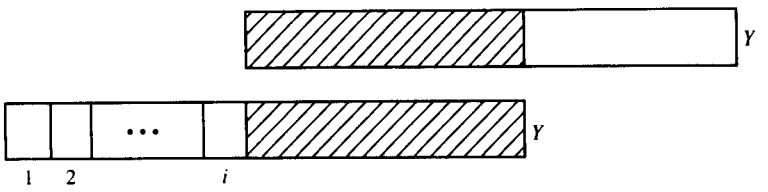


FIGURE 7.1
Two copies of the same string Y, shifted by i positions. The prefix $Y(1 : i)$ is a period if the overlapping portions are identical.

EXAMPLE 7.2:

Let $Y = ababababa$. Then, the prefix $U = abab$ is a period of Y, since $Y = U^2a$. String Y also has the following periods: ab, $ababab$, $abababab$. □

The period of a string Y is the shortest period of Y. Let p be the length (also called size) of the period of Y. We will also refer to p as the period of Y. Hence, p is the smallest integer between 1 and m such that $Y(i) = Y(i + p)$, for all i satisfying $1 \leq i \leq m - p$.

The string Y is called **periodic** if its period p satisfies $p \leq m/2$.

EXAMPLE 7.3:

Consider the following two strings: $Y = abcaabcab$ and $Z = abcabcab$. String Y is not periodic, since its shortest period is $abcaabc$; string Z is periodic, and has the period $p = 3$. □

An important fact about the periods of a string is stated in the next lemma. Before giving the lemma we review a simple version of Euclid's algorithm for computing the greatest common divisor δ of two positive integers p and q.

Remark 7.1: To compute the greatest common divisor δ of p and q, we set $\delta := p = q$ if $p = q$. Otherwise, we assume without loss of generality that $p > q$. Then, it is easy to verify that $\delta = gcd(p - q, q)$. Therefore, computing δ is now reduced to finding the greatest common divisor of the two integers $p' = p - q$ and $q' = q$. Clearly, $p' + q' < p + q$. Hence, the process can be continued until the two remaining integers are equal. We then set δ to their common value. □

Lemma 7.1 (Periodicity Lemma): *If a string Y has two periods of sizes p and q, and $|Y| \geq p + q$, then Y has a period of size $gcd(p, q)$, where $gcd(p, q)$ is the greatest common divisor of p and q.*

Proof of Lemma 7.1: If $p = q$, the proof follows trivially. Let $p > q$ and let $|Y| = m$.

Claim: Y has a period of size $p - q$.

Proof of Claim: Since Y has a period of size p, we have that $Y(i) = Y(i + p)$, for $1 \leq i \leq m - p$. This fact implies $Y(i - q) = Y(i + p - q)$, for $q + 1 \leq i \leq m - (p - q)$. But Y also has a period of size q. Hence, $Y(i) = Y(i - q) = Y(i + p - q)$, for $q + 1 \leq i \leq m - (p - q)$. On the other hand, $Y(i) = Y(i + p) = Y(i + p - q)$, for $1 \leq i \leq q$. Note that we have used here the fact that $m \geq p + q$. Hence, $Y(i) = Y(i + p - q)$, for $1 \leq i \leq m - (p - q)$, and Y has a period of size $p - q$.

The rest of the proof follows from Euclid's algorithm for computing the *gcd* of p and q (Remark 7.1). □

Corollary 7.1: *Let p be the period of string Y of length m. If Y has any period of size q such that $q \leq m - p$, then q is a multiple of p.* □

Remark 7.2: Corollary 7.1 applies if $p + q \leq m$. On the other hand, we should emphasize that Y may have a period q greater than $m - p$ that is *not* a multiple of p. The string $Y = aabcaabcaa$ of period $p = 4$ has also a period of size 9. □

The following lemma is crucial to the parallel string-matching algorithm presented in the next section.

Lemma 7.2: *Let Y be a string of length m whose period is p, and let Z be an arbitrary string of length $n \geq m$. Then, the following two statements hold.*

1. *If Y occurs in Z at positions i and j, then $|i - j| \geq p$.*
2. *If Y occurs in Z at positions i and $i + d$, where $d \leq m - p$, then d must be a multiple of p. If $0 < d \leq \frac{m}{2}$, then Y must also occur at positions $i + kp$, where k is an integer such that $kp \leq d$.*

Proof: Suppose that Y occurs in Z at positions i and j—say, $i < j \leq i + m$. Then, $Y(1 : j - i)$ is a period of Y. Hence, $j - i \geq p$, where p is the period of Y. This result establishes this first statement of the lemma.

Concerning the second statement, note that, if Y occurs at positions i and $i + d$, then $Y(1 : d)$ is a period of Y, and hence d must be a multiple of p, since $d \leq m - p$ (by Corollary 7.1). If $1 \leq d \leq \frac{m}{2}$, then clearly Y is periodic and Y will appear in all positions $i + kp$ such that $kp \leq d$, where k is an integer. □

Corollary 7.2: *Let Y be a string whose period is p, and let Z be an arbitrary string of length n. Then, Y can occur in Z at most n/p times.* □

7.1.2 THE WITNESS ARRAY

We now introduce a function on strings that will play a fundamental role in our parallel string-matching algorithm.

Let Y be a string of length m and period p. Let $\pi(Y) = \min\left(p, \lceil \frac{m}{2} \rceil\right)$. That is to say, $\pi(Y)$ is equal to the period p if Y is periodic; otherwise, $\pi(Y)$ is equal to $\lceil \frac{m}{2} \rceil$. A **witness function** ϕ_{wit} is defined as follows: (1) $\phi_{wit}(1) = 0$, and, (2) for i between 2 and $\pi(Y)$, $\phi_{wit}(i) = k$, where k is some index for which $Y(k) \neq Y(i + k - 1)$.

Intuitively, imagine that a copy of Y is placed on top of Y such that the first and the ith positions are aligned. Then, the overlapping portions of the two copies cannot be identical, because otherwise $Y(1 : i - 1)$ would be a period of Y (recall that $i \leq \pi(Y) \leq p$, the period of Y). Hence, there exists at least one position for which the corresponding symbols are different. The index k is one such position.

A witness function will be represented by an array $WITNESS(1 : r)$ in the obvious way, where $r = \pi(Y)$.

EXAMPLE 7.4:

Consider the two strings $Y = abcaabcab$ and $Z = abcabcab$, introduced in Example 7.3. The following are two possible *WITNESS* arrays corresponding to Y and Z:

$$Y : WITNESS = (0, 3, 2, 2, 5)$$
$$Z : WITNESS = (0, 3, 2)$$ □

Given a string Z of length $n \geq m$, consider the problem of determining all the positions where the string Y can occur in Z. The array *WITNESS* corresponding to Y provides a powerful tool to disqualify many positions of Z as possible candidates for a matching.

Let i and j be two arbitrary positions of Z such that $|i - j| < \pi(Y)$. We know, from Lemma 7.2, that Y cannot occur at positions i and j simultaneously. Let us place two copies of Y on top of Z, one starting at position i, the other starting at position j (see Fig. 7.2). Then, $WITNESS(j - i + 1)$ provides a position at which the two copies of Y differ. Hence, a simple test comparing the symbols at these locations and the symbol at the corresponding location of Z will eliminate at least one of the positions i and j for a possible occurrence of Y in Z.

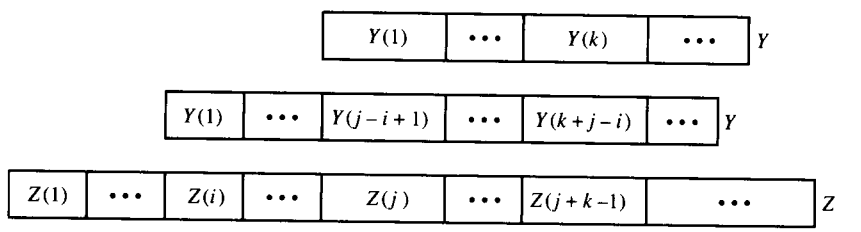


FIGURE 7.2
Elimination of one of the indices i or j as possible candidates for a matching by using $WITNESS(j - i + 1) = k$. Since $Y(k) \neq Y(k + j - i)$, comparing $Z(j + k - 1)$ with $Y(k)$ and $Y(k + j - i)$ will eliminate at least one of the indices i and j as possible locations where Y can occur in Z.

The algorithm is given more formally next.

ALGORITHM 7.1
(Duel(i, j))

Input: *(1) A string Z of length n; (2) a WITNESS array of another string Y of length $m \leq n$; and (3) two indices i and j, where $1 \leq i < j \leq n - m$, such that $j - i < \pi(Y) = \min\left(p, \lceil \frac{m}{2} \rceil\right)$ and p is the period of Y.*

Output: *One of i or j; string Y cannot occur in Z at the eliminated position.*

begin

1. *Set k: = $WITNESS(j - i + 1)$*
2. **if** $Z(j + k - 1) \neq Y(k)$ **then return**(i)
   else return(j)

end

EXAMPLE 7.5:

Let $Y = abcabcab$ with the following *WITNESS* array: $WITNESS = (0, 3, 2)$, and let $Z = abcaabcabaa$. Consider the two positions $i = 5$ and $j = 7$ of Z. Since $WITNESS(7 - 5 + 1) = 2$, the duel algorithm checks $Y(2) = b$ against $Z(8) = a$. Since $Y(2) \neq Z(8)$, the index $i = 5$ is returned. □

Lemma 7.3: *Let Y and Z be two given strings, and let $i < j$ be two distinct indices of Z such that $j - i < \pi(Y)$. Then, no matching of Y in Z can occur at the position eliminated by duel(i, j). Moreover, this procedure takes $O(1)$ sequential time.*

Proof: Since $2 \leq j - i + 1 \leq p$, we have that $WITNESS(j - i + 1) = k \neq 0$. This fact implies that $Y(k) \neq Y(k + j - i)$. Consider location $j + k - 1$ of Z. We cannot simultaneously have the two equalities $Z(j + k - 1) = Y(k)$ and $Z(j + k - 1) = Y(k + j - i)$ (see Fig. 7.2). Clearly, if $Y(k) \neq Z(j + k - 1)$, then Y cannot occur at location j in Z. Hence, index i is returned. If $Z(j + k - 1) = Y(k)$, then j is returned, since $Z(j + k - 1) \neq Y(k + j - i)$, and hence Y cannot occur at location i of Z. In either case, no matching occurs at the eliminated index. Note that the index returned does not necessarily represent a matching at that location.

The duel algorithm can be implemented trivially in $O(1)$ sequential time; hence, the lemma follows. □

7.1.3 *THE PERIODS OF THE PREFIXES

In this section, we address the following string problem. Let Z and Y be strings of lengths n and m, respectively, such that $n \geq m$. Let i be an arbitrary position of Z such that Y does not occur in Z at postion i, where $1 \leq i \leq n - m + 1$. Let j be the smallest index such that $Y(j) \neq Z(i + j - 1)$. Such an index j always exists because Y does not occur in Z at position i (see Fig. 7.3). Our problem is to determine the closest next position (after i) that should be considered for the possible occurrence of Y in Z.

At first thought, it may seem that the location $i + 1$ is the next possible candidate position. However, based on the information we have, it may be possible to eliminate many of the consecutive positions $i + 1, i + 2, \cdots, n - m + 1$ from consideration. We next justify this statement more rigorously.

Assume that the index j is greater than 1. Then, we know that the two substrings $Y(1 : j - 1)$ and $Z(i : i + j - 2)$ are identical, since j is the smallest index such that $Y(j) \neq Z(i + j - 1)$ (see Fig. 7.3). Hence, the substring $Y(1 : j - 1)$ has occurred at location i in Z. By Lemma 7.2, the next possible occurrence of $Y(1 : j - 1)$ in Z is at location $i + D(j)$, where $D(j)$ is the period of $Y(1 : j - 1)$. It follows that Y cannot occur in Z at any of the locations $i + 1, i + 2, \cdots, i + D(j) - 1$. Therefore, the next possible location of Z where Y can possibly occur is $i + D(j)$.

The case when $j = 1$ provides no useful information; hence, the next candidate position is $i + 1$, which can be expressed as $i + D(j)$ as in the case where $j > 1$.

EXAMPLE 7.6:

Consider the strings $Y = abcabcab$ and $Z = abcaabcabaa$. Let us try to match Y at location $i = 1$ of Z. Location 5 is the first position of a mismatch; hence, $j = 5$. The period of the prefix $Y(1 : 4) = abca$ is equal to 3; thus, $D(5) = 3$.

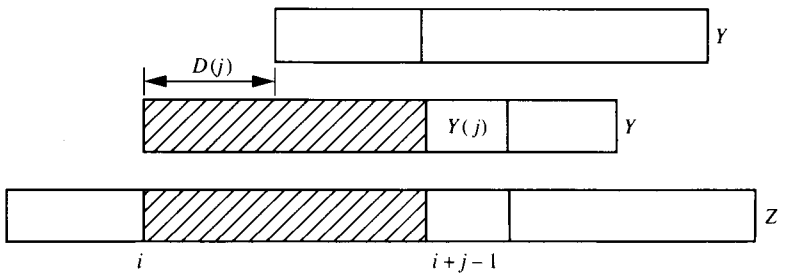


FIGURE 7.3
Determination of the next closest position where Y can possibly occur in Z. The index j is the smallest index such that $Y(j) \neq Z(i + j - 1)$; hence the two substrings $Y(1 : j - 1)$ (lower copy) and $Z(i : i + j - 2)$ are identical (hatched portions). Let $D(j)$ be the period of the substring $Y(1 : j - 1)$. The next possible location where $Y(1 : j - 1)$ can occur is $i + D(j)$; therefore, this location is the next location that should be considered for a possible occurrence of Y in Z.

This result implies that the next possible occurrence of Y in Z is at location $i + D(j) = 4$; hence, there is no need to consider positions 2 and 3 for a possible match. □

The function D introduced previously plays a fundamental role in the linear-time sequential algorithm for string matching described in Section 7.3.4.

7.2 String Matching

Given a *text* string and a *pattern* string, the **string-matching problem** is to find all the occurrences of the pattern in the string. More precisely, let the text and the pattern be represented by the arrays $T(1 : n)$ and $P(1 : m)$, respectively, such that $n \geq m$. We wish to determine a Boolean array $MATCH$ such that, for each $1 \leq i \leq n - m + 1$, $MATCH(i) = 1$ if and only if $P(1 : m)$ occurs in T at position i; that is, $P(j) = T(i + j - 1)$, for $1 \leq j \leq m$.

String matching is a fundamental problem in text editing. There are two classic $O(n + m)$ sequential algorithms for string matching: the Knuth-Morris-Pratt (KMP) algorithm, and the Boyre-Moore (BM) algorithm. A simple version of the KMP algorithm will be briefly outlined in Section 7.3.4. The reader can consult the references cited at the end of this chapter for

more details concerning these two algorithms and several extensions. The techniques used in these two algorithms do not lead to an efficient parallel approach for string matching. In Sections 7.3 and 7.4, we develop different techniques that will yield an $O(\log m)$ time parallel string-matching algorithm using $O(n + m)$ operations.

There is a well-known paradigm to handle various pattern-matching problems. It consists of the following two main steps:

1. **Pattern analysis:** This step involves the processing of the pattern only. Some information regarding the structure of the pattern is extracted, and is stored in one or more tables.
2. **Text analysis:** This phase involves the processing of the text using the pattern and the information generated during the pattern-analysis phase.

Our method for handling the string-matching problem will also follow this paradigm. The pattern-analysis phase will consist of generating the array *WITNESS*, as introduced in the previous section; the second phase will generate the output array *MATCH* efficiently, by using the array *WITNESS*. We start by describing the latter phase.

7.3 Text Analysis

Our input to the text-analysis phase consists of the text array $T(1 : n)$, the pattern array $P(1 : m)$, and the *WITNESS* array of the pattern $WITNESS(1 : r)$, where $r = \pi(P) = \min(p, \lceil \frac{m}{2} \rceil)$ and p is the period of the pattern P. Recall that $WITNESS(i) = k$ implies that $P(k) \neq P(i + k - 1)$, for $2 \leq i \leq r$ and $1 \leq k \leq m - i + 1$. We shall describe the algorithm to find all the occurrences of the pattern P in the text T.

The brute-force algorithm would try to match $P(1 : m)$ against $T(i : i + m - 1)$ for each position i, where $1 \leq i \leq n - m + 1$. Clearly, $\Theta(m)$ operations are required for each such position, leading to an algorithm that uses a total of $O(nm)$ operations. This algorithm can be implemented in $O(1)$ time on the common CRCW PRAM, since the m comparisons required for each position i can be done concurrently in one parallel step, and the value $MATCH(i)$ is nothing but the Boolean AND of the m outcomes.

In spite of its speed, the brute-force algorithm requires an excessive number of operations compared with the linear number of operations used by several of the known sequential algorithms. We develop an alternative strategy that reduces the total number of operations to $O(n)$.

7.3.1 THE NONPERIODIC CASE

We start by considering the case when P is nonperiodic. The corresponding *WITNESS* array is of size $\lceil \frac{m}{2} \rceil$. In addition, we know from Lemma 7.2 that, given any segment of the text T of length $\leq \frac{m}{2}$, P can occur at most once in that segment. This observation suggests the following approach.

Decompose T into blocks T_i, each T_i consisting of $\lceil \frac{m}{2} \rceil$ consecutive characters. Each such block can contain at most one position where a matching can occur. Suppose that we have eliminated all but one possible match point in each T_i. Then, the brute-force algorithm can be applied to all the possible match points concurrently in $O(1)$ time, using a total of $O\left(\frac{2n}{m} \times m\right) = O(n)$ operations. We now proceed to show how to disqualify all the positions but one, in each block T_i.

Initially, all the positions of a block T_i are potential candidates. Let j and k be two arbitrary such positions. The function *duel*(j, k) introduced in Algorithm 7.1 will eliminate one of the positions j and k in $O(1)$ sequential time using the array *WITNESS*. Therefore, the *duel* function can be used to eliminate all but one position as follows. We construct a balanced binary tree on the elements of T_i such that the value stored at each leaf is the index of the corresponding element, and the value stored at an internal node v is the value returned when the *duel* function is applied to the values stored in the children nodes. The index stored at the root is the only possible candidate for a matching. This process takes $O(\log m)$ time, using $O(m)$ operations.

Our text-analysis algorithm for the case when the pattern in nonperiodic is stated more formally next.

ALGORITHM 7.2

(Text Analysis, Nonperiodic Pattern)

Input: *Three arrays* $T(1:n)$, $P(1:m)$, *and* $WITNESS\left(1:\frac{m}{2}\right)$ *corresponding to a text, a pattern and a witness function of the pattern, respectively. It is assumed that P is nonperiodic, and m is even.*

Output: *The array MATCH, indicating all the positions where the pattern occurs in the text.*

begin

1. Partition T into $\lceil \frac{2n}{m} \rceil$ *blocks* T_i, *each block containing no more than* $\frac{m}{2}$ *consecutive characters.*

2. For each block T_i, *eliminate all but one position as a possible candidate for a matching by using a balanced binary tree, where each internal node u contains the index returned by the function duel*(i, j), *and i and j are the indices stored in the children of u.*

3. For each candidate position, verify whether P occurs at that position in T by using the brute-force algorithm.

end

EXAMPLE 7.7:

Let $T = babaababaaba$ and $P = abaab$. Clearly, P is nonperiodic and $\pi(P) = \lceil 5/2 \rceil = 3$. Suppose we are given the array $WITNESS = (0, 1, 2)$. Algorithm 7.2 starts by decomposing T into four blocks, each block containing three consecutive characters. Let $T_1 = bab$, $T_2 = aab$, $T_3 = aba$, and $T_4 = aba$. All these blocks are then handled concurrently. After the first round of duels, we obtain $duel(1, 2) = 2$, $duel(4, 5) = 5$, $duel(7, 8) = 7$ and $duel(10, 11) = 10$. The outcomes of these duels are based on $WITNESS(2) = 1$. After the second round of duels, we get $duel(2, 3) = 2$, $duel(5, 6) = 5$, $duel(7, 9) = 7$ and $duel(10, 12) = 10$. Note that the outcomes of $duel(7, 9)$ and $duel(10, 12)$ are based on $WITNESS(3) = 2$ (a comparison between $P(2) = b$ and the nonexistent character $T(13)$ is assumed to yield a true value to the inequality $P(2) \neq T(13)$). Therefore we are left with the potential candidates 2, 5, 7 and 10—one per block. We can easily check that P occurs at locations 2 and 7. □

We now state the following lemma concerning Algorithm 7.2, whose proof is left to Exercise 7.8.

Lemma 7.4: *Algorithm 7.2 correctly identifies all the positions where the nonperiodic pattern can occur in the text. It can be implemented to run in $O(\log m)$ time, using $O(n)$ operations.* □

PRAM Model: Since the information stored in the *WITNESS* array will be used for all the blocks T_i, a concurrent-read capability is required by Algorithm 7.2. No concurrent write is used. Hence, Algorithm 7.2 runs on the CREW PRAM model. □

Remark 7.3: We developed, in Section 2.6, an optimal $O(\log \log n)$ time algorithm to compute the maximum of n numbers. It turns out that the *duel* function behaves like the maximum function, thereby allowing us to use the strategy of Section 2.6 for designing an $O(\log \log m)$ time optimal algorithm to eliminate all but one candidate in each block of size $m/2$. However, the resulting algorithm will require the common CRCW PRAM. The details are left to Exercise 7.9. □

7.3.2 THE PERIODIC CASE

We now consider the case when the pattern is periodic—say, $P(1 : m) = u^k v$, where v is a proper prefix of u (which could be empty) and $|u| = p$ is the period of P. Our *WITNESS* array is then of length p.

Consider the prefix $P(1 : 2p - 1)$ as a new pattern whose occurrences in T are to be found. We can easily adjust the array *WITNESS* so that it corresponds to $P(1 : 2p - 1)$ by using the following lemma.

Lemma 7.5: *Given a pattern $P(1 : m)$ that has period $p < m/2$, the array $WITNESS(1 : p)$ can be assumed to satisfy $WITNESS(j) \leq p$, for any index j.*

Proof: Let $k = WITNESS(j)$. Define $k' = k \bmod p$, if k is not a multiple of p; otherwise, define $k' = p$. Since $P(1 : m)$ is periodic of period p, $P(k) = P(k')$ and $P(j + k - 1) = P(j + k' - 1)$. Hence, we can set $WITNESS(j) := k' \leq p$. □

Since $P(1 : 2p - 1)$ is nonperiodic, we can use Algorithm 7.2 to determine all the occurrences of $P(1 : 2p - 1)$ in the text T in $O(\log p)$ time using $O(n)$ operations. As a result, we know each position i of T at which $P(1 : 2p - 1)$ occurs. Each such position i can be easily checked for an occurrence of u^2v. Now the problem of generating $MATCH(i)$ comes down to checking whether u^2v occurs at positions $i, i + p, \cdots, i + (k - 2)p$, where k is the integer defined before by $P(1 : m) = u^k v$.

The text analysis for the case when the pattern is periodic is given next.

ALGORITHM 7.3

(Text Analysis, Periodic Case)

Input: *Three arrays $T(1 : n)$, $P(1 : m)$, and $WITNESS(1 : p)$ representing respectively a text, a pattern, and a WITNESS array corresponding to $P(1 : 2p - 1)$. We also know that the pattern is periodic and has period p.*

Output: *The array MATCH identifying all the positions in the text where the pattern occurs.*

begin

1. Use Algorithm 7.2 to determine the occurrences of the prefix $P(1 : 2p - 1)$ in T.

2. For each occurrence of $P(1 : 2p - 1)$ in T—say, at position i—find whether $u^2 v$ occurs at i, where $u = P(1 : p)$, $v = P(kp + 1 : m)$, and $k = \lfloor m/p \rfloor$. If it does, set $M(i) := 1$. For all the other positions i such that $1 \leq i \leq n$, set $M(i) := 0$.

3. For each i such that $1 \leq i \leq p$, let S_i be the subarray of M consisting of the bits $(M(i), M(i + p), M(i + 2p), \ldots)$. For each position l of S_i that contains a 1, set $C(i, l) := 1$ if there are at least $k - 1$ consecutive 1's starting at this position. For all the remaining positions l of S_i, set $C(i, l) := 0$.

4. For each $1 \leq j \leq n - m + 1$, let $j = i + lp$, where $1 \leq i \leq p$ and $l \geq 0$. Then, set $MATCH(j) := C(i, l + 1)$.

end

EXAMPLE 7.8:

Let $T = bababababababaab = (ba)^6(ab)^2$, and let $P = abababa = (ab)^3a$. Hence, the pattern P has the period ab of length 2. Since $P(1:3) = aba$, step 1 of Algorithm 7.3 identifies the locations 2, 4, 6, 8, 10, and 13, where $P(1:3)$ occurs in T. The only occurrences that can be extended to $(ab)^2a$ are in locations 2, 4, 6, and 8. Hence, the array M is given by $M = (0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0)$. During the execution of step 3, we generate the two subarrays $S_1 = (0, 0, 0, 0, 0, 0, 0, 0)$ and $S_2 = (1, 1, 1, 1, 0, 0, 0, 0)$. We then look for the occurrence of two consecutive 1's in each of S_1 and S_2. We therefore obtain $C(1, l) = 0$, for all values of $1 \leq l \leq 8$, and $C(2, 1) = C(2, 2) = C(2, 3) = 1$, and $C(2, l) = 0$ for $4 \leq l \leq 8$. Step 4 sets $MATCH(2) := C(2, 1) = 1$, $MATCH(4) := C(2, 2) = 1$, $MATCH(6) := C(2, 3) = 1$, and $MATCH(i) := 0$ otherwise. Therefore, P occurs in T at positions 2, 4, and 6. □

Lemma 7.6: *Algorithm 7.3 determines all the occurrences of the periodic pattern $P(1 : m)$ in the text $T(1 : n)$ in $O(\log m)$ time, using a total of $O(n)$ operations.*

Proof: The correctness proof follows from the discussion before the statement of the algorithm.

The complexity bounds can be estimated as follows. By Lemma 7.4, step 1 takes $O(\log m)$ time, using a total of $O(n)$ operations. As for step 2, testing whether the occurrence of $P(1 : 2p - 1)$ can be extended to $u^2 v$ can be done in $O(\log m)$ time, using $O(p)$ operations, for each such position by the brute-force algorithm. Since $P(1 : 2p - 1)$ can occur at most $O(\frac{n}{p})$ positions in T (Corollary 7.2), the total number of operations for step 2 is $O(n)$.

Setting up the S_i arrays required by step 3 can be done in $O(1)$ time, using a total of $O(n)$ operations. To compute $C(i, l)$, we do the following. We partition each S_i into buckets, each bucket containing $k - 1$ consecutive bits. We compute the suffix sums of each maximal sequence of 1's within each pair of consecutive buckets. The boundary of such a maximal sequence of 1's is either a 0 or a boundary of the combined pair of consecutive buckets. We can use the segmented prefix-sums algorithm (see Exercise 2.5) to compute the suffix sums in $O(\log k) = O(\log m)$ time, using a linear number of operations. Then, $C(i, l) = 1$ if and only if the suffix sum corresponding to position l in S_i is greater than or equal to $k - 1$. Hence, step 3 takes $O(\log m)$ time, using a linear number of operations.

Step 4 is straightforward; it takes $O(1)$ time, using $O(n)$ operations. Therefore, the running time of the whole algorithm is $O(\log m)$, and the total number of operations is $O(n)$. □

PRAM Model: Step 1 of Algorithm 7.3 requires the CREW PRAM model. The remaining steps can be executed on the CREW PRAM within the stated bounds. Note that steps 2, 3, and 4 can be implemented in $O(1)$ time optimally on the common CRCW PRAM. Such an implementation is left to Exercise 7.11. □

7.3.3 COMBINING ALGORITHMS

Combining Algorithms 7.2 and 7.3 developed in Sections 7.3.1 and 7.3.2 we obtain the following theorem concerning the parallel complexity of the string-matching problem, given the WITNESS array of the pattern.

Theorem 7.1: *Given a text $T(1:n)$, a pattern $P(1:m)$, and the WITNESS array corresponding to P, we can determine all the occurrences of P in T in $O(\log m)$ time, using a total of $O(n)$ operations. Moreover, we can do so using the CREW PRAM model.* □

Section 7.3.4 is devoted to a brief presentation of a classical linear-time sequential algorithm for string matching. The reader can skip that section without loss of continuity.

7.3.4 *THE KNUTH-MORRIS-PRATT STRING-MATCHING ALGORITHM

It is interesting to compare our parallel algorithm with the classic linear-time algorithms for string matching. An outline of the KMP string matching algorithm is given in this section. As in the case of the parallel algorithm, we concentrate here on the text analysis.

Let $T(1:n)$ and $P(1:m)$ be the given text and pattern. Assume that the function $D(1:m+1)$ introduced in Section 7.1.3 is generated during the pattern-analysis phase. Recall that $D(1) = 1$, and $D(j)$ is the period of the prefix $P(1:j-1)$, for $2 \leq j \leq m+1$. For convenience, we assume that the pattern is extended by a special character—say, $P(m+1) = \#$—that cannot appear in the text. Hence, $P(m+1) \neq T(i)$ for any $1 \leq i \leq n$.

Imagine placing the pattern P over the text T and (while keeping T fixed) sliding P to the right according to a certain rule to be described shortly. We will use two pointers k and j to the current characters $T(k)$ and $P(j)$ being matched. The algorithm maintains the following invariant.

Invariant 7.1: Imagine P is placed over T such that $P(j)$ and $T(k)$ are aligned. Then, the characters $P(1), P(2), \cdots, P(j-1)$ match the corresponding characters of the text. More formally, $P(i) = T(k-j+i)$, where $1 \leq i \leq j-1$. □

Initially, $k = j = 1$ and the invariant holds vacuously. Consider now an arbitrary step of the algorithm. We compare $P(j)$ and $T(k)$. Two cases might arise:

1. $P(j) = T(k)$: We then move the two pointers forward by setting $j: = j + 1$ and $k: = k + 1$. If $j = m + 1$, then we know that a match has been found, and we set $MATCH(k - m): = 1$. The invariant is clearly maintained in this case.
2. $P(j) \neq T(k)$: No match is possible at the current position of the pattern. Hence, we have to slide the pattern to the right. We have already seen, in Section 7.1.3, that we obtain the next possible candidate position by sliding the pattern $D(j)$ places to the right. Hence, we can set $j: = j - D(j)$. It is easy to verify that the invariant holds in this case as well.

In the algorithm given next, we use the **failure function** F defined by $F(j) = j - D(j)$, where $1 \leq j \leq m + 1$.

ALGORITHM 7.4
(KMP String Matching)

Input: *Three arrays* $T(1 : n)$, $P(1 : m + 1)$, *and* $F(1 : m + 1)$ *representing a text, a pattern and the failure function corresponding to P, respectively.* $P(m + 1)$ *is a special character that cannot appear in the text.*

Output: *The array MATCH indicating all the occurrences of* $P(1 : m)$ *in the text.*

begin

1. *Set* $j: = 1, k: = 1$;

2. **while** $k - j \leq n - m$ **do**

$T(k) = P(j)$: {*Set* $k: = k + 1, j: = j + 1$;
if $j = m + 1$ **then** *set* $MATCH(k - m): = 1$}

$T(k) \neq P(j)$: {*Set* $j: = F(j)$;
if $j = 0$ **then** *set* $k: = k + 1, j: = 1$}

end

EXAMPLE 7.9:

Let $P = abab$ and $T = babaababaa$. The failure function corresponding to P is given by $F = (0, 1, 1, 2, 3)$. Initially, $j = 1$ and $k = 1$, and we have the situation depicted in Fig. 7.4(a). Since $T(1) \neq P(1)$, we set $j: = F(1) = 0$; hence, we advance the text pointer ($k = 2$) and set the pattern pointer j to 1. Both pointers are advanced for the next three iterations because the corresponding characters match. At this point, we are in the situation depicted in Fig. 7.4(b). Now, since $P(4) \neq T(5)$, we set $j: = F(4) = 2$, which is illustrated

in Fig. 7.4(c). Again $P(2) \neq T(5)$; hence, the value is reset to $j := F(2) = 1$. Fig. 7.4(d) illustrates the current relative positions of the strings. The two pointers are then advanced for the next four positions, resulting in $j = 5$ and $k = 9$. Hence, $MATCH(5)$ is set to 1. We next obtain $j := F(5) = 3$, and we continue with $k = 9$. □

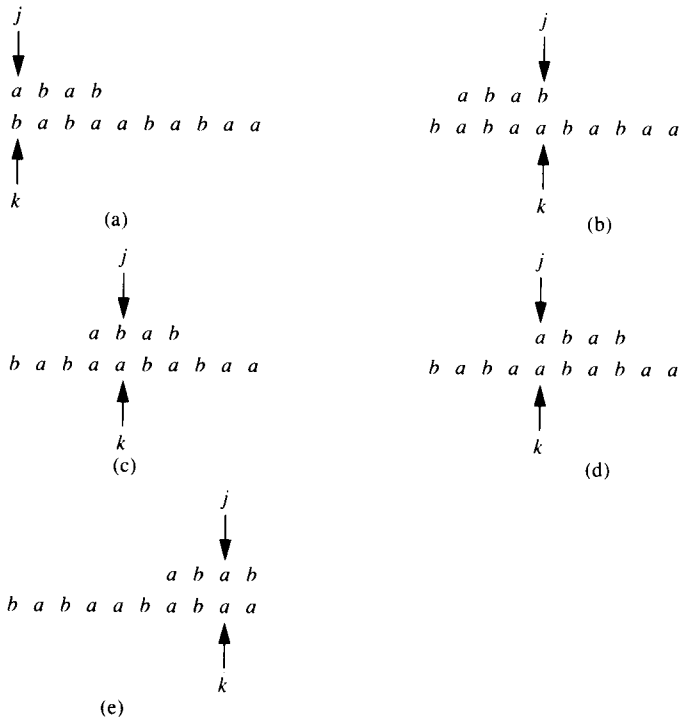


FIGURE 7.4
Several stages of the KMP algorithm. (a) The initial configuration. Since $T(1) \neq P(1)$, the text pointer is advanced; hence, $k = 2$, and the pattern pointer j is set to 1. These pointers will be advanced in the next three iterations, since the corresponding characters match. During the fourth iteration, we notice that $T(5) \neq P(4)$ as shown in (b); hence, the pattern is slid two positions to the right, since the period of *aba* is 2, resulting in the configuration shown in (c). The pattern is then slid one position to the right, since the corresponding characters do not match, resulting in the configuration shown in (d) at the end of the sixth iteration. The two pointers are then advanced four positions; hence, a match will be reported. The pattern is then slid two positions to the right, as indicated in the configuration (e).

Theorem 7.2: *The KMP algorithm correctly identifies all the locations where the pattern P occurs in the text T, in $O(n)$ sequential time.*

Proof: The correctness proof follows from the discussion prior to the statement of the algorithm. As for the running time, notice that, during each iteration of the **while** loop, either both pointers j and k are advanced one position to the right, or the pattern is slid a number of positions to the right; hence, the algorithm terminates after $O(n)$ sequential steps. □

7.4 Pattern Analysis

The two string-matching algorithms presented in Section 7.3 made critical use of precomputed information regarding the pattern. The problem of generating this information from the input pattern is referred to as *pattern analysis*. Our main goal in this section is to present parallel algorithms for computing the array *WITNESS* associated with a given pattern that was used in the parallel string-matching algorithm. The computation of the failure function needed for the KMP algorithm will be outlined in Section 7.4.3.

Recall that the array *WITNESS* was defined for the indices i satisfying $1 \leq i \leq r$, where $r = \min\left(p, \lceil \frac{m}{2} \rceil\right)$ and p is the period of the pattern. During our pattern analysis, we may compute the *WITNESS* values of some indices outside this range, since the period of the pattern has not yet been determined. Clearly, $WITNESS(i) = k \neq 0$ implies that $P(i + k - 1) \neq P(k)$; hence, a nonzero $WITNESS(i)$ provides a concrete example illustrating that the prefix $P(1 : i - 1)$ is not a period of the pattern.

A brute-force algorithm to compute the *WITNESS* array consists of (1) trying to match $P(1 : m - i + 1)$ and $P(i : m)$, for each i such that $2 \leq i \leq \lceil \frac{m}{2} \rceil$, and (2) storing the index of a mismatch, whenever there is one, in $WITNESS(i)$. The algorithm to identify the *smallest* such index, whenever it exists, can be implemented in $O(1)$ time on the common CRCW PRAM, using a total of $O(m)$ operations for each index i, as explained by the next remark.

Remark 7.4: Given an index i, where $2 \leq i \leq \lceil \frac{m}{2} \rceil$, we can determine $WITNESS(i)$ in $O(1)$ time, using $O(m)$ operations, on the common CRCW PRAM as follows. For each k such that $1 \leq k \leq m - i + 1$, set $b_k := 1$ if $P(k) \neq P(k + i - 1)$; otherwise, set $b_k := 0$. We can find the *smallest* index k such that $b_k = 1$ in $O(1)$ time using $O(m)$ operations, on the common CRCW PRAM (see Exercise 2.13). Therefore, determining $WITNESS(i)$ can be done within these bounds. Note that this method ensures that

$WITNESS(i) = k \neq 0$ is the smallest k such that $P(k) \neq P(k + i - 1)$. On the CREW PRAM, this algorithm takes $O(\log m)$ time, using a linear number of operations, for each index i. □

It follows that the brute-force algorithm can be used to determine the *WITNESS* array of the pattern $P(1 : m)$ in $O(1)$ time, using a total of $O(m^2)$ operations on the common CRCW PRAM. We describe in Sections 7.4.1 and 7.4.2 a different algorithm that runs in $O(\log m)$ time, using only $O(m)$ operations. We start with a simple version of this algorithm.

7.4.1 A SIMPLE $O(\log m)$ TIME ALGORITHM TO COMPUTE THE WITNESS ARRAY

Our initial task is to establish a key observation that is similar to the observation that gave rise to the notion of a duel, introduced in Section 7.1.2. It essentially states that, once the subarray $WITNESS(2 : t)$ is computed, for some $t < \min\{p, m/2\}$, where p is the period of the pattern P, then we can compute the *WITNESS* value of at least one of any pair of indices i and j satisfying $|j - i| < t$. This fact is stated more precisely in the next lemma.

Lemma 7.7: *Suppose that we are given* $WITNESS(2 : t)$ *of the string* $P(1 : m)$, *for some* $t < \min\left(p, \frac{m}{2}\right)$, *where* p *is the period of* P. *Then, given any pair of distinct indices* i *and* j, *such that* $t < i, j \leq \lceil \frac{m}{2} \rceil$ *and* $|j - i| < t$, *we can compute at least one of* $WITNESS(i)$ *or* $WITNESS(j)$ *in* $O(1)$ *sequential time.*

Proof: Without loss of generality, assume that $j > i$. Let $k = WITNESS(j - i + 1)$. Note such an index k is available, since $2 \leq j - i + 1 \leq t$, and k must be different from 0 since $t < p$. Hence, $P(k) \neq P(k + j - i)$. We distinguish between the following two cases:

- *Case 1:* $k \leq m - j + 1$. This case is illustrated in Fig. 7.5(a). The character $P(k + j - 1)$ must be different from at least one of $P(k)$ or $P(k + j - i)$. If $P(k + j - 1) \neq P(k)$, then we can set $WITNESS(j) := k$. Similarly, if $P(k + j - 1) \neq P(k + j - i)$, then we can set $WITNESS(i) := k + j - i$.
- *Case 2:* $k > m - j + 1$. This case is illustrated in Fig. 7.5(b). Consider the character $P(k - i + 1)$. Since $k > m - j + 1$, we have that $k - i + 1 > m - j - i + 2$, which implies that $k - i + 1 \geq 1$, since $i, j \leq \lceil m/2 \rceil$. Clearly, $P(k - i + 1)$ must be different from at least one of the two characters $P(k)$ or $P(k + j - i)$. We set $WITNESS(j) := k - i + 1$ whenever $P(k - i + 1) \neq P(k + j - i)$, and $WITNESS(i) := k - i + 1$ whenever $P(k - i + 1) \neq P(k)$. □

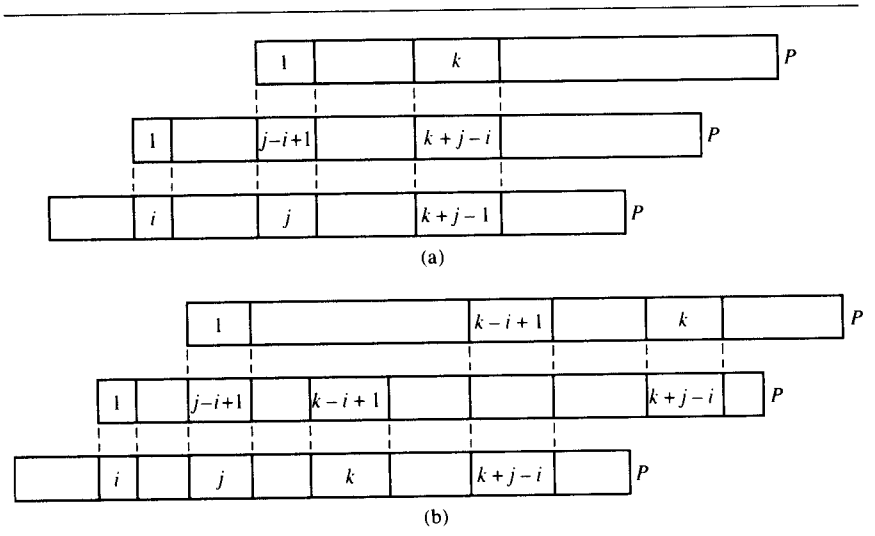


FIGURE 7.5
Determination of *WITNESS*(i) or *WITNESS*(j) using *WITNESS*($j - i + 1$) = k. (a) The first case is where $k \leq m - j + 1$. Since $P(k) \neq P(k + j - i)$, either (1) $P(k) \neq P(k + j - 1)$, and hence we set *WITNESS*(j): = k, or (2) $P(k + j - i) \neq P(k + j - 1)$, and hence we set *WITNESS*(i): = $k + j - i$. (b) The second case is where $k > m - j + 1$. Since $P(k) \neq P(k + j - i)$, either (1) $P(k - i + 1) \neq P(k)$, in which case we can set *WITNESS*(i): = $k - i + 1$, or (2) $P(k - i + 1) \neq P(k + j - i)$, and hence *WITNESS*($j$): = $k - i + 1$.

This process of deducing at least one of *WITNESS*(i) or *WITNESS*(j) from *WITNESS*($j - i + 1$) will be referred to as a **duel** between $P(i)$ and $P(j)$ (or simply a duel between indices i and j). More precisely, the duel function applied to the pair $P(i)$ and $P(j)$ in this section assigns a nonzero value to at least one of *WITNESS*(i) or *WITNESS*(j) and returns the other index, if any, whose *WITNESS* value is still 0.

The following corollary will be used in establishing the correctness of the optimal pattern-analysis algorithm presented in Section 7.4.2.

Corollary 7.3: *Given any pair of distinct indices i and j, such that $t < i, j \leq \lceil \frac{m}{2} \rceil$ and $|j - i| < t$, any computed WITNESS value after a duel between $P(i)$ and $P(j)$ is no larger than WITNESS*($|j - i| + 1$) + $|j - i|$. □

We can use a simple strategy to compute the *WITNESS* array based on the fact shown in Lemma 7.7. The main idea is to start by partitioning the pattern P into blocks, each of size 2. We compute *WITNESS*(2) using the

brute-force algorithm. A duel between the pair of elements in each of the remaining blocks yields a *WITNESS* value to at least one element from each block. We then consider blocks of size 4. The first block $P(1:4)$ has at most one element whose *WITNESS* value has not been determined. Each of the remaining blocks contains at most two elements whose *WITNESS* values have not yet been determined. Hence, we compute the *WITNESS* value of the remaining element in the first block using the brute-force algorithm, and we then apply duels between each pair of the surviving elements in each remaining block. We can continue in this fashion until our *WITNESS* array has been determined. The details are given next.

Assume that $m = 2^s$; the case when m is not a power of 2 can be solved by a simple extension of the case treated here. Let the ***i*-blocks** of P be defined to be the subarrays $P(1:2^i), P(2^i + 1 : 2^{i+1}), \cdots, P(l2^i + 1 : (l+1)2^i)$, for $l + 1 \leq 2^{s-i-1}$ (and hence $(l+1)2^i \leq m/2$). The algorithm consists of at most $s - 1$ iterations and maintains the following invariant.

Invariant 7.2: On entering the ith iteration, the first i-block has at most one index j, where $2 \leq j \leq 2^i$, such that $WITNESS(j) = 0$. Each of the remaining i-blocks has at most two indices k and l such that $WITNESS(k) = WITNESS(l) = 0$. □

Initially, we have $i = 1$, and each block contains two elements. We start by computing $WITNESS(2)$ using the brute-force algorithm. If $WITNESS(2) = 0$, the pattern is then periodic, with period 1, and the algorithm terminates. Otherwise, the *WITNESS* value of at least one element of each of the remaining 1-blocks can be computed using the duel strategy. This finishes the description of the first iteration. The algorithm proceeds to the second iteration.

Suppose the algorithm enters the ith iteration for $i > 1$. The first ith block—that is, the block $P(1:2^i)$—has only one index j such that $WITNESS(j) = 0$ and $j \neq 1$. We call such an index j a **candidate.** We compute $WITNESS(j)$ using the brute-force algorithm. Now, if $WITNESS(j) = 0$, then the pattern is periodic, with period $p = j - 1$. We can stop here, because the array $WITNESS(1:p)$ has been determined completely. Otherwise, we have that $WITNESS(t) \neq 0$, for any $2 \leq t \leq 2^i$. Each of the remaining ith blocks contains at most two candidates (that is, two indices k and l such that $WITNESS(k) = WITNESS(l) = 0$). We can then apply the duel procedure to every such pair. Each i-block is then left with at most one candidate. We now proceed to iteration $i + 1$.

The overall algorithm is described next.

ALGORITHM 7.5
(Simple Pattern Analysis)
Input: *A pattern* $P(1:m)$, *where* $m = 2^s$.

Output: *The array* $WITNESS(1 : r)$, *where* $r = \min(p, 2^{s-1})$, *and* p *is the period of* P.
begin
1. **for** $1 \leq i \leq m/2$ **pardo**
Set $WITNESS(i)$: $= 0$
2. **for** $i = 1$ **to** $s - 1$ **do**
2.1. Let $j \neq 1$ *be the candidate in the first i-block. Compute* $WITNESS(j)$ *using the brute force algorithm.*
2.2. **if** $WITNESS(j) = 0$ **then exit**
2.3. **for** *all i-blocks B (except the first)* **pardo**
Apply $duel(k, s)$, *where* k *and* s *are the candidates in B.*
end

Theorem 7.3: *Algorithm 7.5 correctly generates the array* WITNESS *in* $O(\log m)$ *time, using a total of* $O(m \log m)$ *operations.*

Proof: It is easy to verify that Invariant 7.2 holds whenever the algorithm enters the ith iteration. If the algorithm terminates after the $(s - 1)$st iteration, then, using Invariant 7.2, the array $WITNESS(2 : 2^{s-1})$ has been determined. Otherwise, the algorithm terminates at a certain iteration i such that, after application of the brute-force algorithm, $WITNESS(j)$ remains equal to 0, for some $2 \leq j \leq 2^i$, and $WITNESS(l) \neq 0$ for $l \neq j$ and $2 \leq l \leq 2^i$. This result implies that $j - 1$ is the period of $P(1 : m)$, and the array $WITNESS(1 : j - 1)$ is the desired output.

As for the complexity bounds, note that there are at most $s - 1$ iterations of the **for** loop at step 2. We assume that, after a duel between the pair of indices i and j, the surviving index, if any, is stored in a specific location designated for the corresponding block. Hence, during the next iteration, such a candidate can be accessed in $O(1)$ sequential time. Step 2.1 takes $O(1)$ time, using $O(m)$ operations. A duel applied to a pair of indices takes $O(1)$ sequential time. Hence, step 2.3 takes $O(1)$ time, using $O\left(\frac{m}{2^i}\right)$ operations. Therefore, each iteration can be done in $O(1)$ time, using $O(m)$ operations. Since there are $O(\log m)$ iterations, the claimed complexity bounds follow. □

EXAMPLE 7.10:

Consider the pattern $P = abcabcab$; hence, $m = 8$, in this case. The algorithm has at most two iterations. During the first iteration, $WITNESS(2)$ is determined to be equal to 1. We then perform a duel between $P(3)$ and $P(4)$. In this case, we obtain that $WITNESS(3) = 2$. We now proceed to the second iteration. The candidate in the first 2-block is $P(4)$. Computing $WITNESS(4)$, we obtain that $WITNESS(4) = 0$. Hence, P has a period of size 3 and we stop here. □

PRAM Model: Step 2.1 requires a concurrent write of the same value, if it is to be executed in $O(1)$ time. Step 2.3 requires a concurrent read, since two different i-blocks may need the same *WITNESS* value. However, this substep does not require any concurrent-write instructions. Therefore, the common CRCW PRAM is needed to achieve the stated complexity bounds. □

7.4.2 AN OPTIMAL O(LOG M) TIME ALGORITHM FOR PATTERN ANALYSIS

The main source of inefficiency in Algorithm 7.5 lies in the process of computing the value $WITNESS(j)$ in step 2.1 using the brute-force algorithm. During the ith iteration, the number of operations required by the remaining steps is $O\left(\frac{m}{2^i}\right)$, and thus $O(\Sigma m/2^i) = O(m)$ operations are sufficient to execute these steps over all the iterations. We now describe a more refined method for computing $WITNESS(j)$, for j in the first i-block, possibly over several rounds if necessary. This method will allow us to reduce the total number of operations to $O(m)$.

Recall that, at the beginning of the ith iteration, j is the only index satisfying $2 \leq j \leq 2^i$ and $WITNESS(j) = 0$. Consider computing $WITNESS(j)$ by using the prefix $P(1 : 2^{i+1})$ as the pattern (instead of $P(1 : m)$). Hence, we wish to find an index k, where $1 \leq k \leq 2^{i+1}$, such that $P(k) \neq P(j + k - 1)$ if such an index exists. In this case, the number of operations used is $O(2^i)$, as opposed to the $O(m)$ needed had we used the whole pattern P. If such a scheme works, the correponding total number of operations is $O(\sum_i 2^i) = O(m)$ over all the iterations. Since this approach seems promising, we explore it further.

Suppose that the resulting value of $WITNESS(j)$ is different from zero; that is, there exists a k such that $1 \leq k \leq 2^{i+1}$ and $P(k) \neq P(j + k - 1)$. Then, we have a correct *WITNESS* value for the index j, and we can proceed just as before.

Consider now the case when the value of $WITNESS(j)$ remains 0. The fact that $WITNESS(j) = 0$ for the pattern $P(1 : 2^{i+1})$ implies that $p = j - 1$ is the period of $P(1 : 2^{i+1})$, since j is the smallest such index (as computed by the method outlined in Remark 7.4). However, p may or may not be the period of the whole pattern. If p turns out to be the period of $P(1 : m)$, then we can stop immediately, since $WITNESS(1 : p)$ has already been determined. Otherwise, we have to proceed until at some point $WITNESS(j)$ can be determined.

The following lemma states facts concerning the case when p is a period of some prefix of P, but is not a period of the whole pattern.

Lemma 7.8: *Suppose that p is the period of the prefix $P(1 : 2^\alpha)$, but is not a period of $P(1 : 2^{\alpha+1})$. Let w be the smallest index such that $1 \leq w \leq 2^{\alpha+1} - p$*

and $P(w) \neq P(p + w)$. *Assume that the values* $WITNESS(t)$ *are available for all* $2 \leq t \leq p$ *and that* $WITNESS(t) \leq r$, *for some* $r < 2^{\alpha}$. *Given any index* l *in the range* $p < l \leq 2^{\alpha}$, *then the following statements hold:*

1. *If* $l \neq 1 \bmod p$ *and* $l \leq 2^{\alpha} - r$, then we can set $WITNESS(l) := WITNESS(l')$, *where* $l' = l \bmod p$ *such that* $2 \leq l' \leq p$.
2. *If* $l' = 1 \bmod p$, *say* $l = kp + 1$, *for some integer* $k \geq 2$, *then we can set* $WITNESS(l) := w - (k - 1)p$.

Proof: We start by proving the first statement. Let $\beta = WITNESS(l')$. Then, $P(\beta) \neq P(l' + \beta - 1)$. Since $l \leq 2^{\alpha} - r$, we have that $l + \beta - 1 \leq 2^{\alpha}$, and thus $P(l + \beta - 1) = P(l' + \beta - 1)$ because p is the period of $P(1 : 2^{\alpha})$. Therefore, $P(\beta) \neq P(l + \beta - 1)$ and the first statement follows.

We now consider the second statement. Since w is the smallest index such that $P(w) \neq P(p + w)$, p is the period of $P(1 : p + w - 1)$. Hence, $P(w) = P(w - (k - 1)p)$, which implies that $P(w - (k - 1)p) \neq P(p + w)$. But $p + w = l + (w - (k - 1)p) - 1$; therefore, we can set $WITNESS(l) := w - (k - 1)p$. □

Corollary 7.4: *Assume the same hypothesis as in Lemma 7.8, where* $p, r \leq 2^{\alpha - 1}$. *Then, we can determine* $WITNESS(l) \neq 0$, *for all indices* l *in the range* $p \leq l \leq 2^{\alpha - 1}$, *in* $O(1)$ *parallel steps, using a total of* $O(2^{\alpha})$ *operations.*

Proof: Note that, for l in the range $p < l \leq 2^{\alpha - 1}$, l satisfies the condition $l \leq 2^{\alpha} - r$. Therefore, $WITNESS(l)$ is either equal to $WITNESS(l')$, where $l' = l \bmod p$ and $2 \leq l' \leq p$, or $WITNESS(l)$ is equal to $w - (k - 1)p$, where $l = kp + 1$. Since p, w, and $WITNESS(l')$ are assumed to be known, computing each $WITNESS(l)$ takes $O(1)$ sequential time. □

Based on Lemma 7.8 and Corollary 7.4, we develop an $O(\log m)$ time optimal algorithm to compute the *WITNESS* array. We follow the strategy of the simple pattern-analysis algorithm, with the following modification. At the beginning of the ith iteration, we attempt to determine $WITNESS(j) \neq 0$, where $j \neq 1$ is the only candidate in the first block, by using prefixes of the pattern rather than the pattern itself. Before delving into more details, let us mention that the following invariant will be maintained:

Invariant 7.3: On entering the ith iteration, $WITNESS(l) \leq 2^{i+1}$, for all indices l. □

We start the ith iteration by using the prefix $P(1 : 2^{i+1})$ to determine the value of $WITNESS(j)$. If the resulting value is nonzero, we are done with iteration i, and we proceed to iteration $i + 1$. Otherwise, $p = j - 1$ is the period of $P(1 : 2^{i+1})$. We test whether p is a period of $P(1 : 2^{i+2})$, $P(1 : 2^{i+3})$, $\cdots$,

$P(1 : 2^{\alpha})$, where α is the largest index such that p is a period of $P(1 : 2^{\alpha})$. If $\alpha = \log m$, then p is the period of the whole pattern, and hence we are done, since we have already computed $WITNESS(1 : p)$. Otherwise—that is, $\alpha < \log m$—we can use Lemma 7.11 and its corollary to obtain the *WITNESS* values for all the desired indices l such that $p < l \leq 2^{\alpha-1}$, assuming for now that $\alpha \geq i + 2$. Note that all the corresponding *WITNESS* values will then be different from 0. That is, there will no longer exist an index j such that $2 \leq j \leq 2^{\alpha-1}$ and $WITNESS(j) = 0$. We can then use duels to update the desired *WITNESS* values in each of the remaining $(\alpha - 1)$-blocks, except for possibly one index per block. Clearly, $(\alpha - i)$ rounds of duels between pairs of indices may be required, since each $(\alpha - 1)$-block contains $2^{\alpha-i-1}$ i-blocks, each with at most two candidates. Once the rounds are completed we are ready to begin iteration α. We shall show later that, at this point, $WITNESS(l) \leq 2^{\alpha+1}$ for any index l, and hence Invariant 7.3 is maintained.

There is one case that we have ignored; namely, $\alpha = i + 1$. That is, $p = j - 1$ is the period of $P(1 : 2^{i+1})$, but is not the period of $P(1 : 2^{i+2})$, where j is the candidate from the first i-block. This special case does not require any new ideas beyond those already presented. The fact that p is not a period of $P(1 : 2^{i+2})$ yields a nonzero *WITNESS* value for the candidate in the first i-block. We can then use duels to eliminate at least one of the two candidates in each of the remaining i-blocks. We then proceed to iteration $i + 1$.

In summary, the ith iteration consists of the following steps:

1. Let j be the candidate in the first i-block. Determine $WITNESS(j)$ by using the prefix $P(1 : 2^{i+1})$. If the resulting values of $WITNESS(j)$ are different from 0, proceed as in the simple pattern-analysis algorithm (Algorithm 7.5), and go to iteration $i + 1$. Otherwise, $p = j - 1$ is the period of $P(1 : 2^{i+1})$. Proceed to step 2.
2. Find the largest $\alpha \geq i + 1$ such that p is a period of $P(1 : 2^{\alpha})$, but p is not a period of $P(1 : 2^{\alpha+1})$. If $\alpha = \log m$, then p is the period of the pattern, and we are done (**exit**). Otherwise, proceed to step 3.
3. Use Lemma 7.8 and its corollary to eliminate all the candidates in the first $(\alpha - 1)$-block. Apply $(\alpha - i)$ rounds of duels between pairs of candidates in each of the remaining $(\alpha - 1)$-blocks. Proceed to iteration α.

EXAMPLE 7.11:

Consider the pattern $P = abcabcabcabcabcc = (abc)^5c$. During the first iteration, we use the prefix $P(1 : 4) = abca$ to deduce the value $WITNESS(2) = 1$, followed by duels between the pairs $P(3)$ and $P(4)$, $P(5)$ and $P(6)$, and $P(7)$ and $P(8)$. As a result of these duels, we get $WITNESS = (0, 1, 2, 0, 2, 1, 0, 1)$. The candidates for the second iteration are $P(4)$ and $P(7)$. During the second iteration, the prefix $P(1 : 8) = abcabcab$ is used to update $WITNESS(4)$. But

$P(1:8)$ is periodic, with period 3, and thus $WITNESS(4)$ remains 0. We then check to see whether $P(1:3) = abc$ is a period of $P(1:2^4)$, which is the whole pattern. In this case, we obtain $WITNESS(4) = 13$. Notice that this case corresponds to the case when $\alpha = i + 1$ in the preceding algorithm. Now the first 2-block (that is, $P(1:4)$) contains no candidates, and the remaining 2-block contains only one candidate. Therefore, we go to the third iteration. Since $P(7)$ is the only candidate left, we compute $WITNESS(7) = 10$ using the whole pattern. Therefore, we obtain the *WITNESS* array (0, 1, 2, 13, 2, 1, 10, 1). □

Theorem 7.4: *Given a string $P(1:m)$ above, we can compute the corresponding array WITNESS in $O(\log m)$ time, using a total of above $O(m)$ above operations.*

Proof: We start by establishing the correctness of the modified pattern-analysis algorithm.

Claim: The two Invariants 7.2 and 7.3 hold whenever the algorithm enters iteration i.

Proof of the Claim: The fact that Invariant 7.2 holds whenever the algorithm enters iteration i is easy to verify. The proof regarding Invariant 7.3 is more subtle. The fact that a duel applied to indices i and j results in a *WITNESS* value no larger than $WITNESS(|j - i| + 1) + |j - i|$ (Corollary 7.3) will be used in the analysis given next.

Suppose that the algorithm enters iteration α immediately after iteration i terminates. Clearly $\alpha \geq i + 1$. We consider two cases:

- *Case 1:* $\alpha \geq i + 2$. Hence, $P(1:2^\alpha)$ is periodic. Let p be the period of $P(1:2^\alpha)$. Then $WITNESS(p + 1) \leq 2^{\alpha+1} - p$, since p is not a period of $P(1:2^{\alpha+1})$. Moreover, for l in the first $(\alpha - 1)$-block, the following inequalities hold: (1) $WITNESS(l) \leq 2^{i+1}$, if $l \not\equiv 1 \bmod p$, and (2) $WITNESS(kp + 1) = WITNESS(p + 1) - (k - 1)p \leq 2^{\alpha+1} - kp$. Clearly, Invariant 7.3 holds for the first $(\alpha - 1)$-block when the algorithm enters iteration α.

  Now let $u > v$ be any two candidates in any of the remaining $(\alpha - 1)$-blocks. If $u - v + 1 \not\equiv 1 \bmod p$, the duel applied between $P(u)$ and $P(v)$ will result in a *WITNESS* value no larger than $WITNESS(u - v + 1) + u - v \leq 2^{i+1} + 2^{\alpha-1} \leq 2^{\alpha+1}$. If $u - v + 1 = kp + 1$, for some k, then the duel applied between $P(u)$ and $P(v)$ will result in a *WITNESS* value no larger than $WITNESS(kp + 1) + u - v \leq 2^{\alpha+1} - kp + kp = 2^{\alpha+1}$. Therefore, all computed *WITNESS* values are $\leq 2^{\alpha+1}$ on entry into iteration α.

□ *Case 2:* $\alpha = i + 1$. If $P(1 : 2^{i+1})$ is not periodic, the proof is simple. Hence, suppose that $p < 2^i$ is the period of $P(1 : 2^{i+1})$. As before, $WITNESS(p + 1) \leq 2^{i+2} - p$. Any other *WITNESS* value in the first i-block is $\leq 2^{i+1}$. Hence the first i-block satisfies Invariant 7.3 on entering iteration $i + 1$. Consider any of the remaining i-block. Let $P(u)$ and $P(v)$ be the candidates in that block. Any computed *WITNESS* value resulting after a duel between $P(u)$ and $P(v)$ is $\leq WITNESS(|u - v| + 1) + |u - v|$. If $|u - v| + 1 = p + 1$, then we have the upper bound $2^{i+2} - p + p = 2^{i+2}$. Otherwise, any *WITNESS* value is $\leq 2^{i+1} + 2^i \leq 2^{i+2}$.

We shall now analyze the complexity bounds of the modified pattern-analysis algorithm. The cost associated with an iteration i depends on the next iteration α to be entered by the algorithm.

If $\alpha = i + 1$ (and hence $P(1 : 2^{i+1})$ is not periodic, or its period does not extend to the prefix $P(1 : 2^{i+2})$), the computation of the *WITNESS* value of the candidate in the first i-block takes $O(1)$ time, using $O(2^i)$ operations. Each of the remaining i-blocks requires $O(1)$ sequential time. Since there are $O\left(\frac{m}{2^i}\right)$ such blocks, all the corresponding duels can be done concurrently in $O(1)$ time, using a total of $O\left(\frac{m}{2^i}\right)$ operations. The total cost of the ith iteration is then $O\left(2^i + \frac{m}{2^i}\right)$ operations and $O(1)$ time.

Consider now the case when $\alpha > i + 1$. The complexity bounds of the following three main tasks determine the cost associated with iteration i in this case:

1. The first task is determining that the period p of $P(1 : 2^{i+1})$ extends to $P(1 : 2^{\alpha})$, but not to $P(1 : 2^{\alpha+1})$. Note that we can check whether p extends from $P(1 : 2^r)$ to $P(1 : 2^{r+1})$ in $O(1)$ time, using a total of $O(2^r)$ operations, using the brute-force algorithm. Hence, the whole process can be done in $O(\alpha - i)$ time, using a total of $O(\sum_{r=i}^{\alpha+1} 2^r) = O(2^{\alpha})$ operations.
2. The second task is determining the *WITNESS* values of all the candidates j, where $2 \leq j \leq 2^{\alpha-1}$. There are $O(2^{\alpha-i})$ such candidates. By Corollary 7.4, this task can be done in $O(1)$ time, using a total of $O(2^{\alpha-i})$ operations.
3. The third task is reducing the number of candidates in each remaining $(\alpha - 1)$-block to at most one. Each such block has $O(2^{\alpha-i})$ candidates. Hence, we can perform at most $\alpha - i$ rounds of duels, each of which can be done in $O(1)$ time. Hence, this step requires $O(\alpha - i)$ time, and uses a total of $O\left(\frac{m}{2^{\alpha}} \times 2^{\alpha-i}\right) = O\left(\frac{m}{2^i}\right)$ operations.

It follows from the preceding analysis that the algorithm takes $O(\alpha - i)$ parallel steps, with a total of $O\left(\frac{m}{2^i} + 2^{\alpha}\right)$ operations, to move from iteration i to iteration α.

Therefore, the total running time of the algorithm is $O(\log m)$, and the total number of operations used is $O\left(\sum_i \left(\frac{m}{2^i} + 2^i\right)\right) = O(m)$. □

7.4.3 *COMPUTATION OF THE FAILURE FUNCTION

In this section, we outline how the pattern analysis necessary for the implementation of the KMP algorithm can be done efficiently.

Recall that the text analysis of the KMP algorithm required the availability of the **failure function** F defined as follows: $F(1) = 0$, and $F(j) = j - D(j)$, for $j > 1$, where $D(j)$ is the period of the prefix $P(1 : j - 1)$. It turns out that the KMP algorithm itself can be used to compute F, with the pattern playing the dual role of a text and a pattern at the same time.

Let us start by noting that $F(1) = 0$ and $F(2) = 1$, regardless of the given string. Imagine that a copy of $P(1 : m)$ is placed on top of $P(1 : m)$, such that $P(1)$ and $P(2)$ are aligned as shown in Fig. 7.6. We use j to point to a location of the top copy, and k to point to a location of the bottom copy. Hence, $j = 1$ and $k = 2$, initially. We compare $P(1)$ and $P(2)$. If $P(1) = P(2)$, we deduce that the period of $P(1 : 2)$ is equal to 1, and hence $F(3) = 2$. We now move the two pointers to the next locations; that is, we set $j := j + 1$ and $k := k + 1$. Again, if $P(2) = P(3)$, we can set $F(4) := 3$, since the period of $P(1 : 3)$ is still equal to 1, and we can then advance the pointers to the next locations.

We can clearly continue this process until we encounter the case where $P(j) \neq P(k)$. This case implies that the period of $P(1 : k - 1)$ is no longer the period of $P(1 : k)$. Hence, we have to slide the top copy to the right by the smallest possible number of places so that the overlapping portions of $P(1 : k)$ and the top copy are identical. This problem is similar to the one we confronted during the development of the KMP algorithm. The solution applies to this case as well. That is, we obtain the next possible position by sliding the top copy $D(j)$ positions to the right. Note that we will always have $j < k$, and hence $D(j)$ is known at this point. We then set $j := F(j) = j - D(j)$ and continue as before if $j > 0$ (see Fig. 7.6b). If $j = 0$, then the previous value of j was 1 and $P(1) \neq P(k)$. We conclude that $F(k + 1) = 1$, since the period of $P(1 : k)$ is equal to k; we resume the whole process with $k = k + 1$ and $j = 1$.

The algorithm for determining the failure function is given next.

ALGORITHM 7.6

(Computation of the Failure Function)

Input: *a string* $P(1 : m)$.

Output: *The failure function* $F(k)$, *for all* k *such that* $1 \leq k \leq m + 1$.

begin

1. Set $F(1) := 0, F(2) := 1, k := 2, j := 1$;

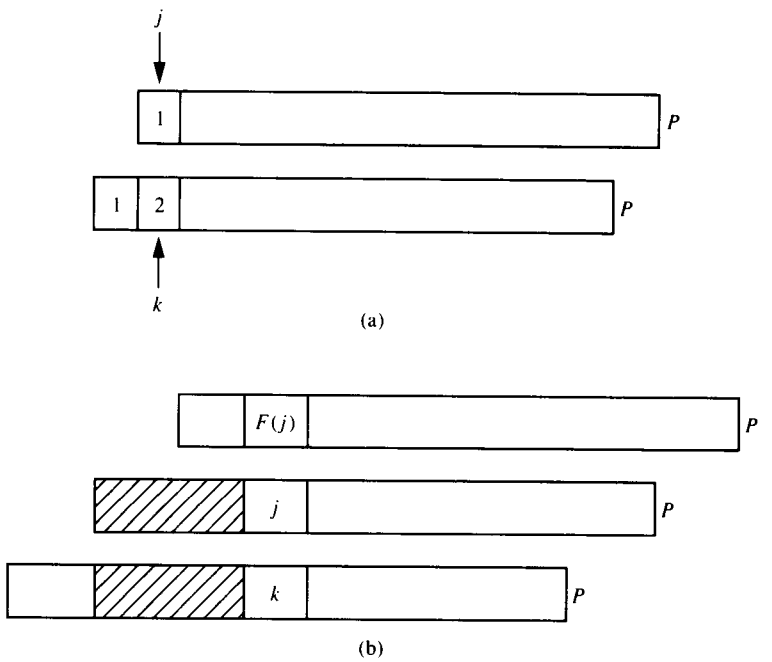


FIGURE 7.6
Determination of the failure function. (a) Initial configuration for computing the failure function. (b) A general step of the computation where the hatched portions are identical. If $P(j) = P(k)$, then we advance the pointers j and k and set $F(k) := j$. Otherwise, $P(j) \neq P(k)$, and hence we slide the top copy to the right so that $F(j)$ and k are aligned, where F is the failure function.

2. **while** $k \leq m$ **do**
$P(k) = P(j)$: {Set $j := j + 1, k := k + 1$;
Set $F(k) := j$}
$P(k) \neq P(j)$: {Set $j := F(j)$;
if $j = 0$ **then**
Set $k := k + 1$ and $j := 1$;
Set $F(k) := 1$}

end

EXAMPLE 7.12:

Consider the same pattern as in Example 7.11—that is, $P = abcabcabcabcc$. Initially, $j = 1$ and $k = 2$. Since $P(1) \neq P(2)$, we obtain $j = F(1) = 0$,

and thus we set $k = 3, j = 1$, and $F(3) = 1$. During the second iteration of the **while** loop, we get $F(4) = 1$, $k = 4$, and $j = 1$ since $P(3) \neq P(1)$. For the following iterations, $P(k)$ will be equal to $P(j)$, as long as $k \leq 15$, since *abc* is the period of $P(1:15)$. We therefore obtain $F(5) = 2, F(6) = 3, F(7) = 4, F(8) = 5, F(9) = 6, F(10) = 7, F(11) = 8, F(12) = 9, F(13) = 10, F(14) = 11, F(15) = 12$ and $F(16) = 13$. We now enter the iteration with $j = 13$ and $k = 16$. Since $P(13) \neq P(16)$, we set $j := F(13) = 10$ and go to a new iteration. We repeat this process, going through the j values of $j = F(10) = 7, j = F(7) = 4, j = F(4) = 1$, and finally $j = F(1) = 0$. In this case, we set $k = 17$, $j = 1$ and $F(17) = 1$, and we exit the **while** loop. □

Theorem 7.5: *Algorithm 7.6 correctly computes the failure function of the string* $P(1:m)$ *in* $O(m)$ *sequential time.*

Proof: The time complexity of the algorithm follows immediately from the observation that, during each iteration, either the top copy slides a number of positions to the right or the two pointers are moved one position to the right.

The correctness follows when we establish the following invariant. At the beginning of any iteration of the **while** loop, the overlapping portions of $P(1:j-1)$ of the top copy and the bottom copy of the pattern are identical; that is, $P(t) = P(t+k-j)$, for $1 \leq t \leq j-1$. The details are left as Exercise 7.17. □

7.5 Suffix Trees

Many string manipulations can be couched within the following framework. Given a text string X, preprocess X and set up appropriate data structures that will allow the fast processing of queries about substrings of X. Some typical queries are the following:

- **On-line string matching:** Does a given string Y occur in X? Related queries are (1) in case Y occurs in X, what is the starting position of the first (or last or all) occurrence(s) of Y in X? and (2) what is the longest prefix of Y which occurs in X? The complexity bounds of processing such queries should depend on $|Y|$ only.
- **Longest repeated substring:** Find the longest substring that occurs in X more than once.
- **Substring identifiers:** Given i, find the shortest substring u of X that identifies position i of X; that is, u occurs in X at location i but u does not occur anywhere else in X. Here, we assume that the last character of X

does not appear anywhere else in X. In this case, the suffix $X(i : n)$ identifies position i of X but a short prefix may be sufficient.

- **Longest common prefix:** Given any two substrings u and v of X, find their longest common prefix.

The *suffix tree* associated with the string X is a data structure that supports all the processing tasks listed, in addition to many other important string operations. Our goal in this section is to show how to set up such a data structure efficiently; in the next section, we describe algorithms for answering some of the queries on our list.

7.5.1 DIGITAL SEARCH TREES

Let us start by considering the **digital search tree** (or **trie**) T associated with a set of distinct strings $Y_1, Y_2, \cdots, Y_m$ over the alphabet Σ. Assume that no Y_i is a prefix of some Y_j, where $j \neq i$. The digital search tree T is a rooted tree with m leaves such that

1. Each edge of T is labeled from the alphabet Σ, and is directed away from the root.
2. No two edges emanating from the same vertex have the same label.
3. Each leaf u is uniquely identified with a string Y_i, in the sense that the concatenation of the labels on the path from the root to u is Y_i.

EXAMPLE 7.13:

Let $\Sigma = \{a, b, c\}$ be the alphabet, and let the given strings be $Y_1 = abaabc$, $Y_2 = baabc$, $Y_3 = aabc$, $Y_4 = abc$, $Y_5 = bc$, and $Y_6 = c$. The digital search tree corresponding to these strings is shown in Fig. 7.7. Each leaf is associated with a string Y_i. Hence, we have a one-to-one correspondence between the leaves and the distinct strings. □

We now return to our original problem—namely, setting up a data structure for a given text string X of length n. We assume that the last character of X is a delimiter #, which does not appear anywhere else in X.

Let T be the digital search tree associated with the n suffixes of X. Then, clearly, T has exactly n leaves, each leaf is uniquely associated with a suffix of X. Tree T captures the similarities and the differences between the suffixes of X. The paths from the root to any two leaves share the portion corresponding to the longest common prefix of the suffixes associated with the two leaves.

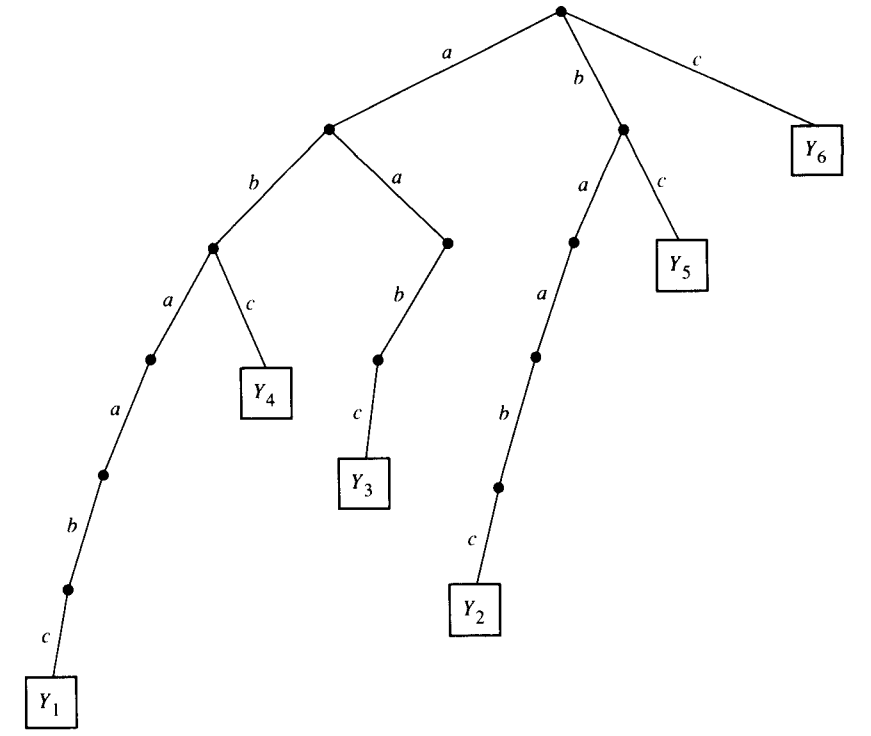


FIGURE 7.7
A digital search tree corresponding to the strings $Y_1 = abaabc$, $Y_2 = baabc$, $Y_3 = aabc$, $Y_4 = abc$, $Y_5 = bc$, and $Y_6 = c$. No two edges emanating from the same vertex have the same label.

EXAMPLE 7.14:

Let $X = ababbaaba\#$ be the given string. The digital search tree T associated with the suffixes of X is shown in Fig. 7.8(a). Consider the problem of determining whether or not the string $Y = baba$ occurs in X. We search T top-down, starting from the root r. We take the edge out of r with the label b, followed by the edge labeled a, and then the edge labeled b. At this point, there is no outgoing edge with the label a. Therefore, we have to stop here and to report that Y does not occur in X. Notice that we have also discovered that bab is the longest prefix of Y that occurs in X. □

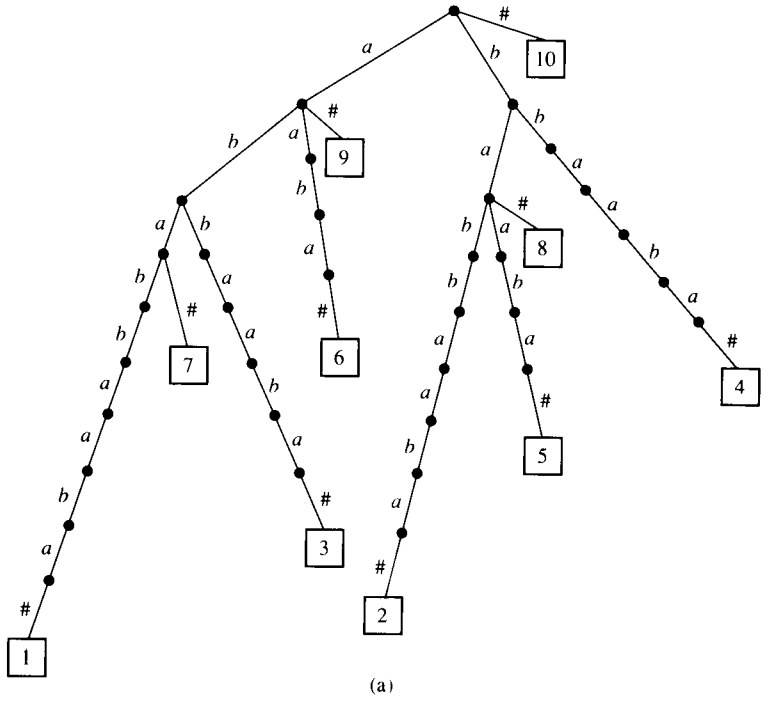


FIGURE 7.8
Digital search tree and associated suffix tree for Examples 7.14 and 7.15.
(a) The digital search tree associated with the string *ababbaaba#*.

7.5.2 SUFFIX TREES

The digital search tree T introduced above may seem to be our desired data structure. However T can be very large. The tree corresponding to the string $X = a^n b^n \#$, for example, will require $\Omega(n^2)$ vertices, which is clearly undesirable. No string of length n will require more than $O(n^2)$ vertices since the length of the path from the root to an arbitrary leaf is $O(n)$ and there are n leaves.

The **suffix tree** associated with X, denoted for the remainder of this chapter by T_X, is a compact version of the digital search tree T. All internal vertices of outdegree 1 are removed, and the label of an edge is now allowed to be a substring of X. More formally, T_X is a digital search tree with n leaves and no nodes of outdegree 1 such that

1. Each edge is labeled with a substring of X.

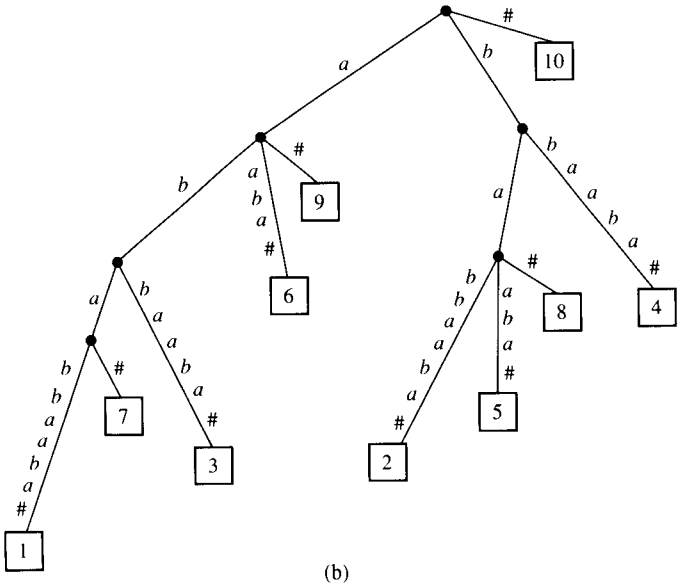
2. No two edges emanating from a vertex are labeled with strings having a (nonempty) common prefix.
3. Each leaf u is associated with exactly one suffix; that is, the concatenation of the labels on the path from the root to u is equal to the suffix.

EXAMPLE 7.15:

Consider the digital search tree shown in Fig. 7.8(a), which corresponds to the string $X = ababbaaba\#$. The resulting suffix tree is shown in Fig. 7.8(b). □

Observe that the suffix tree T_X associated with a string X of length n has at most $2n - 1$ vertices, since the number of internal vertices is bounded by the number of leaves (given that no internal vertex has outdegree 1).

We will denote by $W(v)$ the concatenation of the labels of the path from the root to node v. Clearly, $W(v)$ is a substring of X. Given a substring U of X, the **locus** of U is the node v of T_X such that U is a prefix of $W(v)$, and $W(p(v))$ is a proper prefix of U, where $p(v)$ is the parent of v.



(b)

FIGURE 7.8 (*continued*)
(b) The suffix tree of the string *ababbaaba#*.

7.5.3 ENCODING OF SUBSTRINGS

The crucial property of the suffix tree is to disallow any pair of sibling edges to be labeled by substrings with a common prefix. Our construction of the suffix tree will proceed in $\log n$ stages, where this crucial property is progressively enforced. An important tool used is the efficient encoding of the substrings of X such that identical substrings will have the same encoding.

Given a substring U of X, a **descriptor** of U is a pair $(i, |U|)$ such that the substring of the same length as U starting at position i of X is equal to U. Since the substrings arising in our suffix-tree construction are of length 2^q, for $0 \leq q \leq \log n$, we shall describe a procedure for assigning descriptors to all substrings of X of length 2^q, for $0 \leq q \leq \log n$.

At this point, we have to take a closer look at the input alphabet Σ and the representation of its symbols. The only assumptions we have used in the previous sections are that each character can be stored in a single memory location, and that two characters can be compared in constant time.

The size σ of the alphabet Σ and the representation of the symbols in Σ will play an important role in setting up the suffix tree. A node v in the suffix tree may have up to σ outgoing edges, each of which could be labeled with a different symbol $a \in \Sigma$. In a top-down traversal, we have to proceed from a node v to a child u such that the label of the edge (v, u) begins with a given symbol a. Hence, our data organization should support such a traversal efficiently.

To simplify our presentation, we will make the assumption that $\Sigma \subseteq \{1, 2, \ldots, n\}$, where n is the length of our input string. This assumption can be justified by the fact that we can always sort the symbols appearing in X, and assign to each distinct symbol a unique integer between 1 and n. This preprocessing step will not increase our asymptotic complexity bounds for constructing the suffix tree.

Given this assumption, we represent the children of a node v with an array OUT_v of length n such that $OUT_v(i) = u$ whenever u is the child of v, if it exists, such that symbol i leads from v to u. Since the space required to store OUT_v, for all $v \in T_X$, is $\Theta(n^2)$, we will not initialize these vectors. Hence, the data found in $OUT_v(i)$ may be garbage, and their relevance should be verified.

Our first task is to assign descriptors to some substrings of X. We start by appending X with $n - 1$ instances of the symbol #. This step ensures that a substring of length 2^q, for $0 \leq q \leq \log n$, starting at position i, exists for any i such that $1 \leq i \leq n$. The new string will be denoted by $X\#$.

In the algorithm given next, the array BB (Bulletin Board) is of size $n \times (n + 1)$. Simultaneous write instructions of the form $BB(k, l) := i_1, BB(k, l) := i_2, \cdots, BB(k, l) := i_t$ will result in one of i_j's being stored in location $BB(k, l)$. The choice of i_j is arbitrary. Such a parallel step should be viewed as a way to pass a unique identifier to all processors holding the same (k, l) pair as follows. Processor P_{i_j} with label (k, l) tries to write its index into location $BB(k, l)$. Then,

P_{i_j} reads the content of location $BB(k, l)$. Assuming the arbitrary CRCW PRAM, all the processors holding the same (k, l) pair will read the same value.

ALGORITHM 7.7

(Descriptors of Substrings)

Input: *A string X# such that* $|X| = n = 2^k$, *and the last character of X is the symbol #, which does not appear anywhere else in X. Each X(i) is an integer between 1 and n.*

Output: *An array* $ID[i, q]$, *where* $1 \leq i \leq n$ and $0 \leq q \leq \log n$, *such that* $ID[i, q] = j$ *implies that* $(j, 2^q)$ *is a descriptor of the substring of length* 2^q *starting at location i.*

begin

1. **for** $1 \leq i \leq n$ **pardo**
   Set $BB[X(i),1]: = i$
   Set $ID[i, 0]: = BB[X(i), 1]$
2. **for** $q = 1$ **to** $\log n$ **do**
   for $1 \leq i \leq n$ **pardo**
   2.1. *Set* $k_1: = ID[i, q - 1], k_2: = ID[i + 2^{q-1}, q - 1]$
   2.2. *Set* $BB[k_1, k_2]: = i$
   2.3. *Set* $ID[i, q]: = BB[k_1, k_2]$

end

Remark 7.5: In executing step 2.1 of Algorithm 7.7, the index $i + 2^{q-1}$ may be larger than n, in which case we assign the value $n + 1$ for the nonexistent entry $ID[i + 2^{q-1}, q - 1]$. □

EXAMPLE 7.16:

Let $X = abbabba\#$ be the input string. Represent the symbols a, b and # by the integers 1, 2, and 3, respectively. Assume that, whenever there is a simultaneous write instruction of the form $BB[k, l]: = i$, the smallest such i gets stored in location (k, l) of BB. Then, Algorithm 7.7 generates the following array:

$$ID = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 2 & 3 & 3 & 3 \\ 1 & 1 & 1 & 4 \\ 2 & 2 & 5 & 5 \\ 2 & 3 & 6 & 6 \\ 1 & 7 & 7 & 7 \\ 8 & 8 & 8 & 8 \end{bmatrix}.$$

Consider, for example, the generation of the entry $ID[4, 2]$. During the second iteration and for $i = 4$, we obtain $k_1 = ID[4, 1] = 1$ and $k_2 = ID[6, 1] = 3$. Hence, at step 3.2, we try to set $BB[1, 3] = 4$. At the same time, we also attempt to set $BB[1, 3] = 1$, for $i = 1$. By our convention, the net effect is to set $BB[1, 3] = 1$, and thus $ID[4, 2]$ is set to 1 after execution of step 3.3. Clearly, the substring of length 4 starting at location 4 is *abba*, which is exactly the substring of length 4 starting at location 1. □

Lemma 7.9: *Algorithm 7.7 correctly computes the array ID such that* $ID[i, q]$ *is a descriptor of the substring of X of length* 2^q *starting at location i. The running time of the algorithm is* $O(\log n)$ *and the total number of operations used is* $O(n \log n)$. □

PRAM Model: A concurrent-write capability of the arbitrary type was used in step 2.2 of Algorithm 7.7 to assign a unique identifier for identical substrings. Hence, Algorithm 7.7 runs on the arbitrary CRCW PRAM. □

Remark 7.6: Due to the introduction of the BB array, our algorithm uses $\Theta(n^2)$ space. However, no more than $O(n \log n)$ locations are ever accessed. The total amount of space can be reduced to $O(n^{1+\epsilon})$, for any $0 < \epsilon \leq 1$. The proof is left to Exercise 7.24. □

7.5.4 CONSTRUCTION OF SUFFIX TREES

We are ready to describe the algorithm to construct the suffix tree T_X associated with an input string X, which is assumed to be of length $n = 2^k$, for some integer k.

For each node v of T_X, we keep an array OUT_v representing the outgoing edges of v (in a way to be clarified shortly), a pointer to its parent $p(v)$, and a descriptor denoted by (s_v, l_v) of the substring representing the label of the edge $(p(v), v)$. It follows that the label of the edge $(p(v), v)$ is equal to the substring of X of length l_v starting at position s_v. The array OUT_v is of length n such that $OUT_v(j) = u$ whenever (v, u) is an outgoing edge of v and j is the ID of the label of the edge (v, u). It follows that u is stored in a location of OUT_v that depends on the descriptor of the label of the edge (v, u). The fact that no two sibling edges can have a common prefix ensures that no attempt is made to store two elements into the same location.

An initial approximation to T_X is the tree T_0 of height 1 consisting of a root with n outgoing edges, each edge labeled with a different suffix. Hence, leaf v_i, corresponding to the ith suffix, contains the descriptor (α_i, n), and $OUT_r(\alpha_i) = v_i$, where r is the root and $\alpha_i = ID[i, \log n]$.

The crucial property of allowing no sibling edges to have a common prefix will in general be violated in T_0. We will successively refine T_0 over $\log n$ iterations such that the tree generated at the end of the ith iteration satisfies the following **condition**(i):

1. T_i is a labeled tree with n leaves having no internal vertices with out-degree equal to 1.
2. Each edge (u, v) of T_i is labeled with some substring of X. A descriptor of this substring is stored in node v. In addition, the leaves of T_i are in one-to-one correspondence with the suffixes of X.
3. Let v be an internal vertex of T_i with the outgoing edges (v, u_1), (v, u_2), $\cdots$, (v, u_t). All the prefixes of length $\frac{n}{2^i}$ of the labels of these edges are different. The u_i's are stored in OUT_v as follows: $OUT_v(\alpha_i) = u_i$ if α_i is the *ID* of the prefix of the label of (v, u_i) of length $\frac{n}{2^i}$.

Clearly, T_0 satisfies **condition**(0). We shall describe a general procedure that takes a tree T_{i-1} satisfying **condition**($i - 1$) as input, and generates T_i that will satisfy **condition**(i). The algorithm terminates when $i = \log n$, since $T_{\log n}$ is clearly the suffix tree of X. A high-level description of the procedure is given next. The implementation details will be discussed later.

ALGORITHM 7.8

(Refining T_{i-1} into T_i)

Input: *A tree T_{i-1} satisfying* **condition**($i - 1$). *If $i = 1$, T_0 is the tree of height 1 consisting of a root with n outgoing edges, each edge labeled with a different suffix of a string X of length n.*

Output: *A tree T_i satisfying* **condition**(i). *When $i = \log n$, the tree $T_{\log n}$ is the suffix tree of X.*

begin

for *all internal vertices v of T_{i-1}* **pardo**

1. Let $u_1, u_2, \cdots, u_r$ be the children of v. Partition the u_i's into equivalence classes C_j such that the labels of the edges incoming into the vertices of C_j have the same prefix of length $\frac{n}{2^i}$.

2. For each equivalence class C_j such that $|C_j| > 1$, create a new vertex v'. Make v' the parent of all the vertices in C_j, and make v the parent of v'. Label (v', v) with the common prefix of length $\frac{n}{2^i}$, and adjust the labels of the remaining edges, OUT_v, and set up $OUT_{v'}$.

3. Suppose that v has a single equivalence class that resulted in a new vertex v'. Remove v and make v' the child of the parent $p(v)$ of v. Adjust the label of $(p(v), v')$ and $OUT_{p(v)}$.

end

EXAMPLE 7.17:

Consider the string $X = abbabba\#$ given in Example 7.16. The initial tree T_0 is shown in Fig. 7.9(a). It consists of eight leaves; leaf v_i contains the descriptor $(i, 8)$ representing the ith suffix. During the first iteration, the two vertices v_1 and v_4 are grouped into one equivalence class because they both have the common prefix $abba$ of length $4 = \frac{n}{2}$. Each of the other leaves is in an equivalence class by itself. A new vertex u_1 is created, and is made the parent of v_1 and v_4. The corresponding T_1 is shown in Fig. 7.9(b). During the second iteration, the equivalence classes are given by $\{v_1\}$, $\{v_4\}$, $\{v_2, v_5\}$, $\{v_3, v_6\}$, $\{v_7\}$, and $\{v_8\}$. Clearly, the labels of v_2 and v_5 have the common prefix bb, whereas v_3 and v_6 have the common prefix ba. Hence, two new vertices u_2 and u_3 are created, and the resulting T_2 is shown in Fig. 7.9(c). Finally, during the last iteration, we obtain the equivalence classes $\{v_1\}$, $\{v_4\}$, $\{u_1, v_7\}$, $\{v_2, v_5\}$, $\{u_2, u_3\}$, $\{v_3\}$, $\{v_6\}$, and $\{v_8\}$. Therefore, three new vertices u_4, u_5, and u_6 are created resulting in the tree shown in Fig. 7.9(d). Note that the outdegree of u_2 is now equal to 1, and hence u_2 should be removed. The resulting tree shown in Fig. 7.9(e) is the suffix tree of X. □

Lemma 7.10: *Given that T_{i-1} satisfies* **condition**$(i-1)$*, the tree T_i generated at the end of Algorithm 7.8 satisfies* **condition**(i)*, assuming that the labels and the OUT arrays are computed properly.*

Proof: The only nontrivial part of the proof is to show that step 3 of Algorithm 7.8 does not apply simultaneously to a vertex v and to its parent $p(v)$. Clearly, $p(v)$ cannot be a new vertex introduced in step 2. By **condition**$(i-1)$, $p(v)$ had more than one child. Since $p(v)$ remains the parent of v after step 2, the label of the edge $(p(v), v)$ does not have a common prefix of length $\frac{n}{2^i}$ with any of its sibling edges. Therefore, after the execution of step 2, $p(v)$ is of outdegree larger than 1; hence, step 3 cannot apply to $p(v)$. □

Several important implementation details have to be explained before we can determine the complexity bounds of Algorithm 7.8. Our goal is to allow the implementation of this algorithm in $O(1)$ time, with a total of $O(n)$ operations, which will allow the suffix tree to be constructed in $O(\log n)$ time, using $O(n \log n)$ operations. We discuss the implementation of each step of the algorithm separately.

Implementation of Step 1. The purpose of step 1 is to generate for each u_i an identifier j_{id} to its equivalence class. Let (s_i, l_i) be the descriptor of u_i (a slight abuse of the notation introduced earlier). Hence, the label of the edge (v, u_i) is equal to the substring of X of length l_i starting at location s_i. Let $q = k - i$ (where $n = 2^k$), and hence $2^q = \frac{n}{2^i}$. We set $j_{id} := ID[s_i, q]$. Recall that

$ID[s_i, q]$ is a descriptor of the substring of length 2^q starting at s_i. Hence, all vertices u_i with a common prefix of length 2^q will get the same j_{id} value. Clearly, this step can be implemented in $O(1)$ time, using $O(n)$ operations.

At this point, we recommend that you attempt to solve Exercise 7.27. This exercise will help you to understand the implementation of step 2, described next.

Implementation of Step 2. The purpose of step 2 is to create a new vertex v', which will be the parent of all vertices belonging to the same equivalence class C_j, for each j such that $|C_j| > 1$. This task can be done as follows.

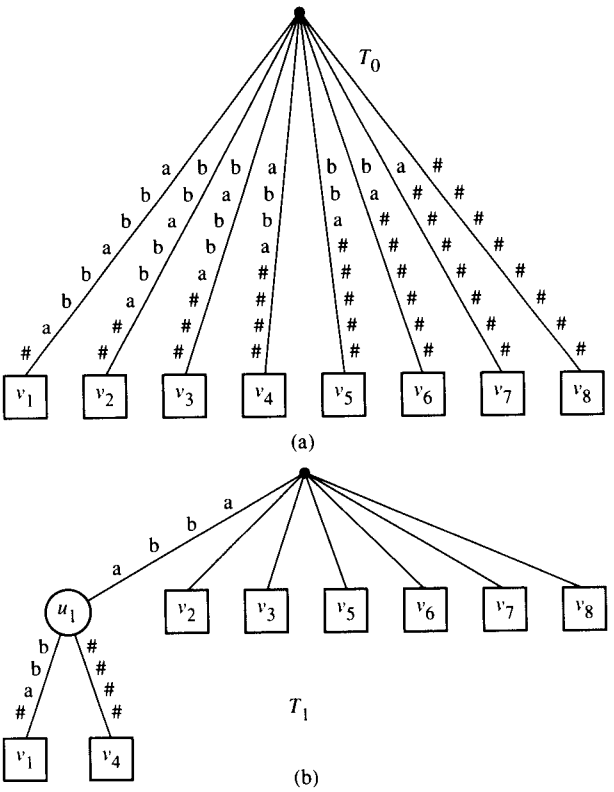
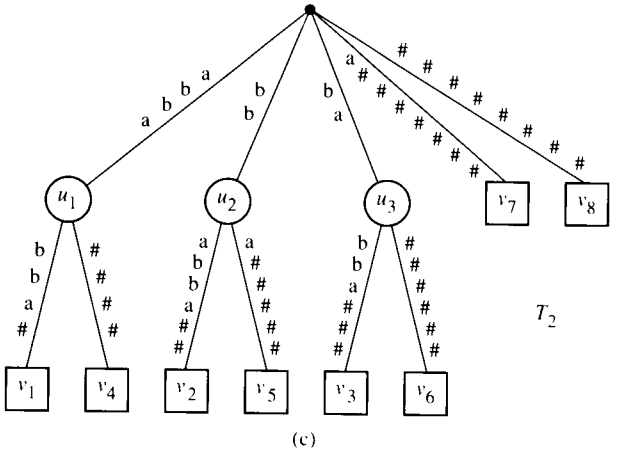
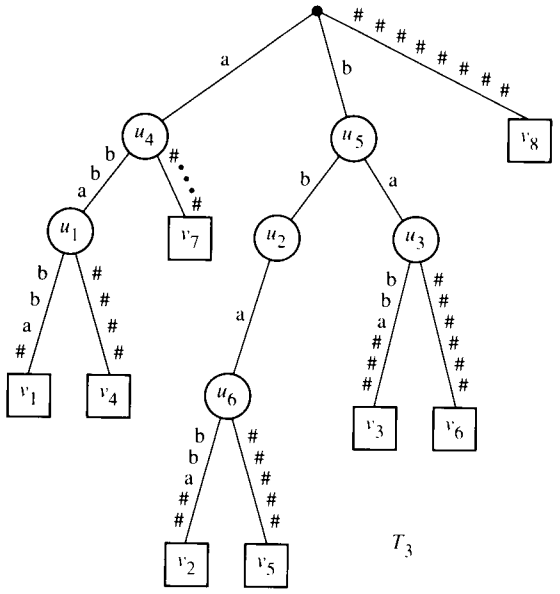


FIGURE 7.9
Generation of a suffix tree by Algorithm 7.8. (a) The initial tree T_0 corresponding to X = *abbabba#*. (b) Tree T_1 generated during the first iteration. The unlabeled edges retain their initial labeling.

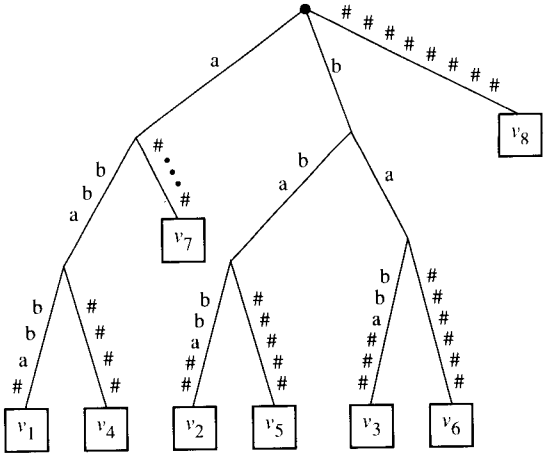


(c)



(d)

FIGURE 7.9 (*continued*)
(c) Intermediate tree T_2 generated during the second iteration. (d) The tree T_3, generated during the third iteration, has a vertex of outgoing degree 1.



(e)

FIGURE 7.9 (*continued*)
(e) The suffix tree of X = *abbabba#* after eliminating the vertex of outgoing degree 1.

We set $OUT_v(j_{id}) := u_i$, for each u_i, where j_{id} is as introduced in the implementation of step 1. Now all vertices u_i, say $u_{i_1}, \ldots, u_{i_t}$, with the same j_{id} value—say, α—will attempt to write into the same location of OUT_v. Since a concurrent write can be detected easily whenever it occurs, we can then set $p(u_{i_m}) := v'$, where v' is a new vertex, for $1 \le m \le t$, and set $p(v') := v$. The descriptor of v' is clearly $(\alpha, 2^q)$, whereas the descriptor of each u_{i_m} is adjusted to $(s_{i_m} + 2^q, l_{i_m} - 2^q)$ (that is, the suffix of the old label starting at location $s_{i_m} + 2^q$). Now we set $OUT_v(\alpha) := v'$, and $OUT_{v'}(j'_{i_m}) = u_{i_m}$, where $j'_{i_m} = ID[s_{i_m} + 2^q, q]$; that is, j'_{i_m} corresponds to the descriptor of the second portion of the prefix of u_{i_m} of length 2^{q+1} (recall that $2^q = \frac{n}{2^i}$, and hence $2^{q-1} = \frac{n}{2^{i-1}}$). No concurrent write occurs here, since no sibling edges are supposed to have a common prefix of length 2^{q+1}. It is not difficult to verify that step 2 can be implemented in $O(1)$ time using $O(n)$ operations.

Implementation of Step 3. The implementation of step 3 is straightforward. Let $p(v) = w$. We make w the parent of v' and adjust OUT_w as follows. Let $j = ID[s_v, q]$; then, set $OUT_w(j) := v'$. The descriptor stored in v' is given by $(s_v, l_v + 2^q)$, where (s_v, l_v) is the descriptor stored in v. Again, this step can be implemented in $O(1)$ time, using a linear number of operations.

Using the preceding observations, we obtain the following theorem.

Theorem 7.6: *We can construct the suffix tree of a string of length n in $O(\log n)$ time, using a total of $O(n \log n)$ operations.* □

PRAM Model: A concurrent-write capability of the arbitrary type was clearly exploited in the implementation of the suffix-tree construction. Hence, Algorithm 7.8 runs on the arbitrary CRCW PRAM. □

Remark 7.7: The algorithm for constructing the suffix tree of a string was given in the work-time presentation framework. The corresponding processor-allocation problem is not straightforward. The details are left to Exercise 7.26. □

7.6 Applications of Suffix Trees

In this section, several applications of suffix trees are sketched. The basic framework is to preprocess a given string X so as to be able to answer queries concerning substrings of X very quickly.

For simplicity, the last character of X is assumed to be #, a delimiter that does not appear anywhere else. We begin with the on-line string-matching problem.

7.6.1 ON-LINE STRING MATCHING

Our problem is to preprocess the string X so that we can quickly answer queries of the following form, "Does a string Y occur in X?"

Let $|Y| = m$. Clearly, each query involving Y requires $\Omega(m)$ operations, since Y is not available for preprocessing, and hence all the characters of Y have to be examined. We aim to handle each such query in $O(\log m)$ time, using $O(m)$ operations, independent of the length of the string X. We shall show next that this goal is indeed possible to attain using suffix trees.

Since X is available for preprocessing, we construct the suffix tree T_X associated with X. During the process of constructing T_X we save the following information:

1. All the $\log n$ arrays BB used in Algorithm 7.7
2. All the intermediate trees $T_0, \cdots, T_{\log n}$, including the vectors OUT_v used to define the list corresponding to the children of v for each T_i

The preprocessing requires $O(\log n)$ time, using a total of $O(n \log n)$ operations, where $|X| = n$.

The processing of on-line string-matching queries begins by assigning names to some substrings of Y that are consistent with the ID names used for the substrings of X. We can do this assignment by using the same BB arrays introduced in Algorithm 7.7.

For $q = 0, 1, \cdots, \lfloor \log m \rfloor$, we assign names to all substrings of Y that are of length 2^q and that begin at positions i, where $i = 1 \bmod (2^q)$ such that $i + 2^q \leq m$. We store the corresponding names in $PID[i, q]$, which is the analog of the array ID introduced for the string X. The total number of substrings of Y involved in this process is $O\left(\sum_{q=0}^{\lfloor \log m \rfloor} \frac{m}{2^q}\right) = O(m)$.

This assignment results in the following important fact. The names of two identical substrings of length 2^q, one from $X\#$ and the other from Y, are equal.

The described initial step of the query processing clearly can be done in $O(\log m)$ time, using a total of $O(m)$ operations.

A sequential algorithm to process the on-line string-matching query proceeds top-down, starting from the root of T_X. We follow the longest path leading to a node v such that v is the locus of Y, if such a node exists. Recall that the locus of Y is the node v of T_X such that Y is a prefix of $W(v)$, where $W(v)$ is the string we obtain by concatenating the labels on the path from the root to v, and $W(p(v))$ is a proper prefix of Y. Clearly, the answer to the query is yes if v exists, and no otherwise. This top-down traversal seems difficult to implement in parallel efficiently. We describe a different method that is reminiscent of binary search.

Our goal is to identify a node $v \in T_X$, if it exists, such that Y is a prefix of $W(v)$, but $W(p(v))$ is a proper prefix of Y (that is, v is the locus of Y in T_X). Clearly, Y occurs in X if and only if such a node v exists. To locate $v \in T_X$, we will determine $O(\log m)$ intermediate nodes in the trees T_i's that will lead to v (if it exists). We begin by figuring out the first node to be explored.

Consider an arbitrary tree T_i, for some $0 \leq i \leq \log n$. The length of the label of each edge of T_i is $\geq \frac{n}{2^i}$. In addition, no sibling edges can have a common prefix of length $\frac{n}{2^i}$.

Let $i_0 = \log n - \lfloor \log m \rfloor$. We know that the prefixes of the edge labels of length $\frac{n}{2^{i_0}} = 2^{\lfloor \log m \rfloor}$ must all be distinct in T_{i_0}. Hence, there exists at most one child u of the root r of T_{i_0} such that $Y(1 : 2^{\lfloor \log m \rfloor})$ is a prefix of the label of (r, u). If no such node exists, then Y cannot appear in X. Otherwise, this node can be identified by $OUT_r(j)$, where $j = PID[1, \lfloor \log m \rfloor]$. The node $u \in T_{i_0}$ will be our first approximation to the desired node $v \in T_X$. Node u and the remaining suffix Y' of Y will be the input parameters for the next step. The details follow:

□ *Step 1.* Let $i_0 = \log n - \lfloor \log m \rfloor$, and let $j = PID[1, \lfloor \log m \rfloor]$. We check the entry $OUT_r(j)$:

1. $OUT_r(j)$ is "empty." Since OUT_r has not been initialized, $OUT_r(j)$ may contain some garbage data. We can check this condition in a

straightforward manner using the prefix $Y(1 : 2^{\lfloor \log m \rfloor})$. If $OUT_v(j)$ is empty we conclude that the prefix $Y(1 : 2^{\lfloor \log m \rfloor})$ does not appear in X. Therefore, the answer to the query is no.

2. $OUT_r(j) = u$. Let (s_u, l_u) be the descriptor of u. If $l_u > 2^{\lfloor \log m \rfloor}$ (and thus $l_u \geq 2^{\lfloor \log m \rfloor + 1}$), then Y appears in X if and only if Y is a prefix of the substring of X of length l_u starting at s_u. In this case, the occurrence of Y can be checked in a straightforward manner in $O(1)$ time, using $O(m)$ operations. Otherwise, $l_u = 2^{\lfloor \log m \rfloor}$, and hence the label of the edge (r, u) is identical to the prefix $Y(1 : 2^{\lfloor \log m \rfloor})$. Node u and the remaining suffix Y' of Y (beginning at position $2^{\lfloor \log m \rfloor} + 1$) will be the input parameters of the next step.

□ *Step 2.* This step takes a node $u \in T_{i_0}$ and Y' as input and finds the desired node $v \in T_X$, if it exists, in at most $\lfloor \log m \rfloor$ iterations.

Iteration q ($q = 1, 2, \ldots, \lfloor \log m \rfloor$). Let $u \in T_{i_0+q-1}$ and Y' be the input parameters to iteration q. The goal is to find whether Y' follows an occurrence of $W(u)$. Consider the tree T_{i_0+q}. There are two possibilities:

1. The node u appears in T_{i_0+q}. There are two subcases that might arise. The first is where $|Y'| < \frac{n}{2^{i_0+q}}$, in which case we do nothing and proceed to the next iteration using the same input parameters. The second subcase is where $|Y'| \geq \frac{n}{2^{i_0+q}}$. Suppose that Y' starts at position p of Y, and set $j := PID[p, i_0 + q]$. We examine $OUT_u(j)$ and proceed as in step 1.
2. The node u does not appear in T_{i_0+q}. Thus, the labels of all the outgoing edges of u in T_{i_0+q-1} had the same prefix—say, z—of length $\frac{n}{2^{i_0+q}}$, which resulted in the creation of a new node u' in T_{i_0+q}. The node u' and the parent $p(u)$ of node u in T_{i_0+q-1} appear in T_{i_0+q}. Again, two subcases might arise. The first subcase is where $|Y'| < \frac{n}{2^{i_0+q}}$. The string Y occurs in X if and only if Y' is a prefix of z. Using the descriptor stored in u', we can check that Y' is a prefix of z with a straightforward algorithm in $O(1)$ time, using $O(m)$ operations. The second subcase occurs where $|Y'| \geq \frac{n}{2^{i_0+q}}$. We can check whether z is a prefix of Y' by using the two arrays ID and PID. A negative answer implies that Y cannot occur in X. Otherwise, we can proceed to the next iteration, using the node u' and the remaining suffix of Y' as the input parameters.

We summarize our results concerning the on-line string-matching problem in the following theorem, whose proof follows from the preceding discussion.

Theorem 7.7: *The on-line string-matching query can be processed in $O(\log m)$ time, using $O(m)$ operations, where the string Y is of length m. The preprocessing algorithm of the text string X takes $O(\log n)$ time, using $O(n \log n)$ operations, where n is the length of X.* □

7.6.2 LONGEST REPEATED SUBSTRING

The problem is to find the longest substring of X that occurs more than once.

The preprocessing algorithm consists of building the suffix tree T_X associated with X, and computing $|W(v)|$ for each node $v \in T_X$. The actual computation of $|W(v)|, v \in T_X$, can be performed with the pointer-jumping technique (Section 2.2). This amount of preprocessing takes $O(\log n)$ time, and uses $O(n \log n)$ operations.

The answer to the longest-repeated-substring query is an internal node u such that $|W(u)| = \max_{v \in T_X} |W(v)|$. Recall that the problem of computing the maximum of n elements can be solved in $O(\log \log n)$ time, using a linear number of operations, on the common CRCW PRAM (Section 2.6). Therefore, the processing of the longest repeated substring query can be performed in $O(\log \log n)$ time, using a linear number of operations.

7.6.3 SUBSTRING IDENTIFIERS

Our goal is to preprocess the string X such that we can answer the following type of queries in constant time: "Given i, determine a substring identifier for position i of X."

We simply construct the suffix tree T_X during the preprocessing phase.

Given i, we can locate the leaf v of T_X associated with the suffix $X(i : n)$ in $O(1)$ sequential time. The answer is the concatenation of $W(p(v))$ and the first character in the label of $(p(v), v)$.

The preprocessing algorithm can be implemented in $O(\log n)$ time, using a total of $O(n \log n)$ operations, whereas a substring identifier query can be answered in $O(1)$ sequential time.

The processing for a query about the **longest common prefix** is left to Exercise 7.29.

7.7 Summary

We have examined in this chapter strings and related basic computational problems. We have shown how to exploit the periodicities of strings to design

efficient parallel algorithms. Our solution to the string-matching problem proceeded in two phases, consisting of a pattern-analysis phase and a text-analysis phase. The duel function was used to disqualify many candidate positions during the text-analysis phase, and to generate the *WITNESS* array during the pattern-analysis phase. As a result, we were able to describe an optimal logarithmic-time parallel algorithm for the string-matching problem. The approach followed is quite different from the one used by the well-known sequential algorithms for string matching.

We have also introduced the suffix tree of a string, which is a powerful data structure that can be used for manipulating substrings of a given string. The construction of the suffix tree of a string of length n can be accomplished in $O(\log n)$ time, using $O(n \log n)$ operations. Several string-processing tasks using suffix trees were used to illustrate the usefulness of this data structure.

A summary of the algorithms described in this chapter are given in Table 7.1. Randomized algorithms for string matching will be described in Chapter 9.

TABLE 7.1
ALGORITHMS PRESENTED IN THIS CHAPTER.

Algorithm		Section	Time	Work	PRAM Model
7.1	*duel*(i, j)	7.1.2	$O(1)$	$O(1)$	EREW
7.2	Text Analysis, Nonperiodic Pattern	7.3.1	$O(\log m)$	$O(n)$	CREW
7.3	Text Analysis, Periodic Case	7.3.2	$O(\log m)$	$O(n)$	CREW
7.4	KMP String Matching	7.3.4	$O(n)$	—	Sequential Algorithm
7.5	Simple Pattern Analysis	7.4.1	$O(\log m)$	$O(m \log m)$	Common CRCW
	Optimal Pattern Analysis	7.4.2	$O(\log m)$	$O(m)$	Common CRCW
7.6	Computation of the Failure Function	7.4.3	$O(m)$	—	Sequential Algorithm
7.7	Descriptors of Substrings	7.5.3	$O(\log n)$	$O(n \log n)$	Arbitrary CRCW
7.8	Refining T_{i-1} into T_i	7.5.4	$O(1)$	$O(n)$	Arbitrary CRCW
	Construction of a Suffix Tree	7.5.4	$O(\log n)$	$O(n \log n)$	Arbitrary CRCW
	On-Line String Matching	7.6.1	$O(\log m)$	$O(m)$	Arbitrary CRCW
	Longest Repeated Substring	7.6.2	$O(\log \log n)$	$O(n)$	Common CRCW
	Substring Identifier	7.6.3	$O(1)$	$O(1)$	EREW

m, length of pattern

n, length of text string

Exercises

7.1. Let X be the string $X = abaababaabaab$ over the alphabet $\Sigma = \{a, b\}$.
- **a.** Determine all the periods of X.
- **b.** Determine a *WITNESS* array for X.
- **c.** For each prefix of X, determine its period.

7.2. **a.** Let $X(1:m)$ be a string with period p. Can the substring $X(1:p)$ be periodic?
- **b.** Suppose that $X = u^k u'$, where $|u| = p$ and $|u'| < p$. Can the substring uu' have a period $< p$?

7.3. Let X and Y be two arbitrary strings, and let p be the period of X. Suppose that X occurs at positions i and j of Y, but X does not occur at positions $i + p$ or $j + p$. Show that $|i - j| > |X| - p$.

7.4. Let $S(1:m)$ be a string, and let $S(1:m')$ be a proper prefix of S. Suppose that p is the period of $S(1:m')$, but p is not a period of $S(1:m)$. Show that any period q of $S(1:m)$ satisfies $q > m' - p$.

7.5. Consider the *Fibonacci strings* defined over the alphabet $\Sigma = \{a, b\}$: $\phi_1 = b, \phi_2 = a$; and $\phi_n = \phi_{n-1}\phi_{n-2}$, for $n \geq 3$. For example, $\phi_3 = ab$, $\phi_4 = aba$, and $\phi_5 = abaab$. Recall that the Fibonacci numbers are defined by $F_n = F_{n-1} + F_{n-2}$, and $F_1 = F_2 = 1$.
- **a.** Show that $\phi_{n-2}\phi_{n-1} = c(\phi_{n-1}\phi_{n-2})$, where $c(\alpha)$ denotes changing the two rightmost characters of the string α. For example, $c(\phi_5) = ababa$.
- **b.** Consider the string $S = \phi_{n+2}$. Show that the period of $S(1:j)$ is F_n, for $F_{n+1} \leq j \leq F_{n+2} - 2$. What is the period of $S(1:F_{n+2} - 1)$? Deduce the period of S.

7.6. The duel algorithm (Algorithm 7.1) is written such that it returns exactly one of the input indices. However, it may be possible to eliminate both input indices. Rewrite the algorithm to eliminate one or two indices whenever possible.

7.7. Let $X(1:m)$ be a given string, and let $\phi(i) = \min\{k \mid X(k) \neq X(i + k - 1)\}$, and $\phi(i) = m - i + 1$, if no such k exists, for $1 \leq i \leq m$.
- **a.** Show that the function D introduced at the end of Section 7.1 is given by $D(j) = \min\{i \mid \phi(i + 1) + i \geq j\}$.
- **b.** Describe an $O(\log m)$ time algorithm to compute $D(j)$, for $1 \leq j \leq m$, given the function ϕ. What is the total number of operations used?

7.8. Establish the correctness of Lemma 7.4 concerning the text-analysis algorithm (Algorithm 7.2) in the case of a nonperiodic pattern.

7.9. Show that text-analysis algorithm for nonperiodic patterns (Algorithm 7.2) can be modified to run in $O(\log \log m)$ time, using a linear number of operations. Specify the PRAM model used.

7.10. **a.** Let M be a Boolean array of size m. Given an integer $k < m$, consider the problem of computing a Boolean array M' of the same size such that $M'(i) = 1$ if and only if $M(i) = M(i + 1) = \cdots = M(i + k) = 1$. Carefully explain how to compute M'. Your algorithm must use $O(m)$ operations. The running time of your algorithm should be $O(\log m)$ for the EREW PRAM model.

b. Develop an $O(\log k)$ time algorithm to compute M' on the EREW PRAM. Your algorithm must use $O(n)$ operations.

7.11. Show that the algorithm to reduce the periodic to the nonperiodic case (Algorithm 7.3) can be implemented in $O(1)$ time on the common CRCW PRAM, if we exclude step 1.

7.12. Given a symbol c from an alphabet Σ, a **scaling** of c by a multiplicative factor k is the string $cc \ldots c$, repeated k times. We denote the resulting string by c^k. The scaling by a factor of k of a string X is the string $X(1)^k X(2)^k \ldots X(n)^k$, where n is the length of X.

Develop an algorithm to identify all the positions in a text string $T(1:n)$, where a match of a pattern string P scaled to k occurs. Your algorithm should run in $O(\log m)$ time, using a linear number of operations. *Hint:* Represent a string by a set of symbols, where no two consecutive symbols are identical, and by a corresponding set of integers.

7.13. Let X be a string of length m. Consider the problem of computing the function $\delta : \{1, 2, \ldots, m\} \rightarrow \{0, 1\}$ such that $\delta(i) = 1$ if and only if $X(i:m) = X(1:m - i + 1)$.

a. Specify all locations for which $\delta(i) = 1$.

b. Develop an optimal parallel algorithm to compute δ. Handle with care the substring $X(m - p:m)$, where p is the period of X. Do not make any assumptions about the alphabet.

7.14. Let X and Y be two strings of length m over an alphabet Σ. Let i be an index such that $X(i:m) = Y(1:m - i + 1)$. What is the smallest $j > i$, if it exists, that satisfies $X(j:m) = Y(1:m - j + 1)$? Given such an index i, develop an optimal parallel algorithm to identify all the indices $j > i$ satisfying $X(j:m) = Y(1:m - j + 1)$ by using the results established in Exercise 7.13.

7.15. Does the simple pattern-analysis algorithm (Algorithm 7.5) generate the smallest k such that $WITNESS(i) = k$ if and only if $P(k) \neq P(i + k - 1)$, whenever $WITNESS(i) \neq 0$? Explain your answer.

7.16. Modify the simple pattern-analysis algorithm (Algorithm 7.5) to handle the case when the length of the pattern is not necessarily a power of 2. Also, explain how to set up the information regarding the candidates in each block so that they can be identified in $O(1)$ sequential time.

7.17. Establish the correctness of the algorithm for the computation of the failure function (Algorithm 7.6).

7.18. The pattern-analysis algorithm described in Section 7.4.2 to generate the array *WITNESS* of a pattern string of length m can be modified to run in $O(\log \log m)$ time.

a. Show how this modification can be achieved. *Hint:* At the ith stage, consider blocks of size $k_i = m^{1-2^{-i}}$. Compute $\leq \frac{k_i}{k_{i-1}}$ witness values in the first k_i-block. Handle the remaining blocks in $O(1)$ time using duels.

b. *Can you make your algorithm optimal? Explain your answer.

7.19. This exercise introduces the notion of a **deterministic sample** of a pattern string P, which characterizes, in a certain sense, the occurrences of P in a text.

Let P be nonperiodic of length m, assumed to be even. Imagine that $\frac{m}{2}$ copies of P, denoted by $P_1, P_2, \ldots, P_{\frac{m}{2}}$, are placed on top of each other, each shifted one position to the right from the copy directly below it (see Fig. 7.10).

a. Show that there exist a copy P_s and a set of indices $\{s_1, s_2, \ldots, s_l\}$, where $l < \log m$, such that, for any $i \neq s$, there is a mismatch between the copies P_i and P_s in at least one of the locations $s_1, s_2, \ldots, s_l$ of P_s. *Hint:* Start by considering P_1 and $P_{\frac{m}{2}}$. Clearly, $WITNESS\left(\frac{m}{2}\right) = k \neq 0$. Look at the column going through locations k of $P_{\frac{m}{2}}$ and $\frac{m}{2} + k - 1$ of P_1. At least one-half of the copies will disagree with one of the two corresponding characters.

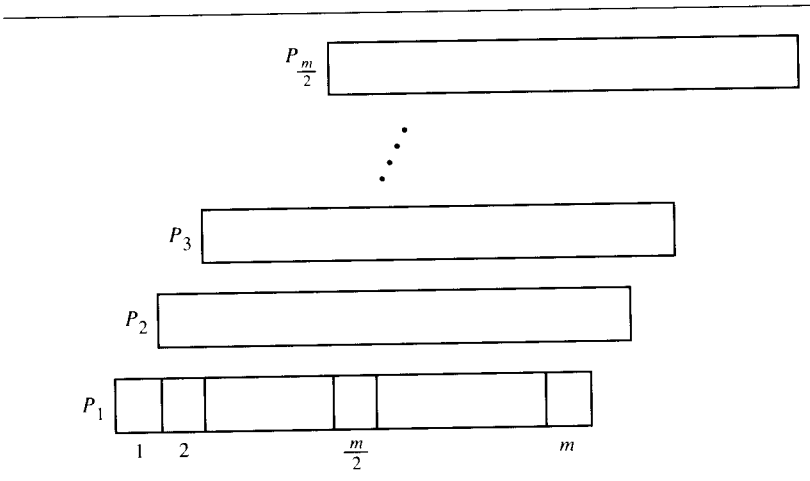


FIGURE 7.10
A layout of the $\frac{m}{2}$ copies of the pattern P for defining a deterministic sample.

b. Develop an $O(\log^2 m)$ time algorithm to determine P_s and the corresponding indices. Your algorithm must be optimal. What is the weakest PRAM model required?

The set of indices $\{s_1, s_2, \ldots, s_l\}$ will be called a **deterministic sample** of P.

7.20. The notion of a deterministic sample of a nonperiodic pattern was introduced in Exercise 7.19. Develop an $O(1)$ time algorithm to find all the occurrences of P in a text $T(1 : n)$ based on this notion. Your algorithm should use $O(n \log m)$ operations.

7.21. **a.** *Refine the algorithm that you developed in Exercise 7.20 to obtain an $O(\log m)$ time optimal algorithm for string matching based on the notion of a deterministic sample.

b. *Can you make your algorithm run in $O(\log^* m)$ time while preserving optimality?

7.22. This exercise requires some understanding of **regular expressions** and their relationship to finite automata. Let α be a regular expression over the alphabet Σ. A classic result shows how to construct a nondeterministic finite automaton M that accepts the language denoted by α such that the number of states of M is no larger than $2|\alpha|$, where $|\alpha|$ is the number—say, n—of symbols in α.

Develop a parallel algorithm to construct M in $O(\log n)$ time, using a linear number of operations. Represent M by a directed graph whose vertices are the states—say, $1, 2, \ldots, m$—and whose arcs define state transitions. Assume that α is given in an array of length n. *Hint:* Start by finding the tree representation of α using Exercise 3.32.

7.23. Consider the following three **edit** operations on a string X: (1) *deleting* a character a from X with cost $D(a)$, (2) *inserting* a character a between two consecutive locations of X at a cost $I(a)$, and (3) *substituting* some occurrence of a of X with an occurrence of b at a cost $S(a, b)$.

The **string-editing problem** for input strings X and Y is to find a minimum-cost sequence of edit operations that will transform X into Y. Let $C(i,j)$ be the minimum cost of transforming $X(1 : i)$ into $Y(1 : j)$, and let $|X| = n$ and $|Y| = m$.

a. Show that

$$C(i,j) = \min \begin{cases} C(i-1, j-1) + S(X(i), Y(j)), \\ C(i-1, j) + D(X(i)), \\ C(i, j-1) + I(Y(j)), \end{cases}$$

for all $1 \le i \le n$, and $1 \le j \le m$.

b. The table $C(n, m)$ can be represented by a graph $G = (V, E)$ such that each entry $C(i, j)$ is represented by a vertex $v_{i,j}$ with three (or fewer) incoming arcs from $v_{i-1,j-1}$, $v_{i-1,j}$ and $v_{i,j-1}$, whenever they

exist. Deduce an $O(\log^2(nm))$ time algorithm to compute the array C. What is the total number of operations used?

c. You can improve the efficiency of the algorithm from part (b) by using a divide-and-conquer strategy to determine the desired shortest path in G. Assuming $n = m$, develop such a parallel algorithm that uses $O(n^2 \log n)$ operations. *Hint:* Consider shortest paths from a vertex $v_{1,i}$ to vertices $v_{n,j}$. Establish that the order of the $v_{n,j}$s implies a similar order on the vertices of the paths determined by a horizontal cut.

7.24. Show how to reduce the space used by Algorithm 7.7 from $O(n^2)$ to $O(n^{1+\epsilon})$, for any fixed $0 < \epsilon \leq 1$. The asymptotic complexity bounds must stay the same. *Hint:* Express k_2 by its representation in the base n^ϵ, assuming n^ϵ to be an integer, in $\frac{1}{\epsilon}$ steps. Use a new BB array of size $n^\epsilon \times (n + 1)$.

7.25. Suppose you are given an algorithm A that computes a one-to-one function F from a set of n numbers in the range $[1, m]$ into the range $[1, 10n]$. Such a function F is called a **perfect hash function.** The space used by the algorithm A is $O(n)$. Show that A can be used to reduce the space of the suffix-tree construction to $O(n)$.

7.26. **a.** Solve the processor allocation problem corresponding to Algorithm 7.7, assuming that there are n processors available.

b. Solve the processor allocation problem corresponding to Algorithm 7.8, assuming that, at the beginning of the ith iteration, a processor is allocated to every edge of T_{i-1}, and that, when the algorithm terminates, a processor must be allocated to every edge of T_i.

7.27. Let $T = (V, E)$ be a rooted tree such that each vertex v is labeled with an integer $l(v) \in \{1, 2, \ldots, q\}$. The tree T is represented by an array F such that $F(v)$ is the parent of v, $1 \leq v \leq n$, and $F(v) = 0$ in the case that v is the root. Consider the problem of transforming T into another tree T' as follows. Let $u_1, u_2, \ldots, u_m$ be the children of a vertex v. Let C_1, $C_2, \ldots, C_t$ be the partition of the u_i's induced by the labeling function For each C_i, create a new vertex w_i, where $1 \leq i \leq t$, and make w_i the parent to all vertices in C_i. Make v the parent of all the w_i's. This process is supposed to be performed on all the vertices of T to obtain T'.

a. Develop an $O(\log n)$ time CREW PRAM algorithm to transform T into T'. What is the space needed by your algorithm?

b. Develop an $O(1)$ time CRCW PRAM algorithm to accomplish the same transformation. Carefully handle the creation of new vertices. What is the space needed? *Hint:* Imagine a processor standing next to each vertex of T.

7.28. Let X be a string, and let T_X be the corresponding suffix tree. Each vertex v contains a descriptor (s_v, l_v) representing the label of the edge directed toward v. You are asked to compute $|W(v)|$, for each vertex v,

where $W(v)$ is the substring obtained by concatenating the labels of the edges on the path from the root to v. Develop an $O(\log n)$ time algorithm to compute these values, where n is the number of vertices in T_X. What is the total number of operations used? Which PRAM model is used?

7.29. **a.** Show that the suffix tree T_x of a string x can be used to answer a longest-common-prefix query: Given two substrings u and v of x, find their longest common prefix. What are the complexity bounds of your algorithm, ignoring the bounds required for preprocessing?

b. Suppose that u and v are suffixes of x. How quickly can you process the longest-common-prefix query?

c. Generalize your algorithms to the case where you are given a set of strings and you wish to handle longest-common-prefix queries on their substrings.

7.30. Modify the on-line string-matching algorithm described in Section 7.6.1 such that it finds all the occurrences of the pattern. What are the corresponding complexity bounds of your algorithm?

7.31. Let X and Y be two strings of length m over the alphabet $\Sigma = \{1, 2, \ldots, m\}$. Let S be the set of the m prefixes of X and the m suffixes of Y.

a. Develop an algorithm to compute the characteristic function $\chi : S \rightarrow \{1, 2, \ldots, 2m\}$ such that two strings are mapped into the same value if and only if they are equal. Your algorithm should run in $O(\log m)$ time. What is the total number of operations used?

b. Use the characteristic function to determine the bit vector $\delta(1 : m)$ such that $\delta(i) = 1$ if and only if $X(i : m) = Y(1 : m - i + 1)$, $1 \leq i \leq m$. This problem is referred to as the **suffix-prefix-matching problem.**

7.32. Develop a linear time-sequential algorithm to solve the suffix-prefix-matching problem introduced in Exercise 7.31, without any assumption on the input alphabet. *Hint:* Use a strategy similar to that used in the KMP string-matching algorithm (Algorithm 7.4).

7.33. Use the algorithm for the suffix-prefix-matching problem introduced in Exercise 7.31 to solve the string-matching problem. The complexity bounds of the resulting algorithm should be dominated by those of the algorithm to solve the suffix-prefix-matching problem.

7.34. Let $T(1 : n)$ and $P(1 : m)$ be a text string and a pattern string, respectively, $m \leq n$. Let S be a substring of T. The *edit distance* $d(S, P)$ between S and P is the minimum number of edit operations required to transform S into P. Edit operations were introduced in Exercise 7.23.

Consider the problem of identifying all the substrings S of T such that $d(S, P) \leq k$, where k is an input parameter. Each edit operation required represents a difference between S and P. This problem is called **string matching with *k*-differences.**

a. Let D be the $(m + 1) \times (n + 1)$ matrix such that $D(i, l)$ is the minimum number of differences between $P(1 : i)$ and any substring of T ending at location l. Show that

$$D(i,l) = \min\{D(i-1,l) + 1, D(i,l-1) + 1, \overline{D}(i-1,l-1)\},$$

where

$$\overline{D}(i-1,l-1) = \begin{cases} D(i-1,l-1) & \text{if } P(i) = T(l), \\ D(i-1,l-1) + 1 & \text{if } P(i) \neq T(l). \end{cases}$$

What is the sequential complexity of the resulting dynamic programming algorithm? Develop a parallel version of this algorithm, and state its complexity bounds.

b. The diagonal d of D consists of all entries $D(i, l)$ such that $l - i = d$. Let L be the matrix such that $L(d, e)$ is the largest row i such that $D(i, l) = e$ and $D(i, l)$ is a diagonal d. (1) Suppose that $L(d + 1, e - 1)$, $L(d - 1, e - 1)$ and $L(d, e - 1)$ are known. Show that $L(d, e) = r + j$, where $r = \max\{L(d + 1, e - 1) + 1, L(d - 1, e - 1), L(d, e - 1) + 1\}$, and j is the largest integer such that $P(r + 1 : r + j) = T(d + r + 1 : d + r + j)$. (2) Deduce an $O(nk + n \log n)$ time sequential algorithm to compute L. *Hint:* Note that the index j defines the largest common prefix of two suffixes. Use a suffix tree that supports LCA queries.

c. Deduce a parallel algorithm to solve the problem of string matching with k-differences in $O(k + \log m)$ time. What is the total number of operations used?

7.35. Let $T(1 : n)$ be a text string, and let $P_i(1 : m)$, $1 \leq i \leq k$, be k patterns. The problem of **multipattern string matching** is to determine, for each position of the text, whether a pattern string occurs there. If a match exists, the index of a matching pattern should be reported. Show that the two-dimensional array-matching problem can be reduced to the multipattern string-matching problem. What are the resulting complexity bounds for the algorithm to solve the two-dimensional array-matching problem?

Bibliographic Notes

Periodicities in strings were first studied in a purely mathematical context. The periodicity lemma (Lemma 7.1) was described by [10, 19]. The two classical linear-time string-matching algorithms of Knuth-Morris-Pratt (KMP) and Boyer-Moore have appeared in [6, 16]. The Boyer-Moore algorithm compares the pattern and a corresponding portion of the text in a right-to-left scan over the pattern (as oppposed to the left-to-right scan of the KMP algorithm), and is on average sublinear. The first optimal parallel string-matching algorithm was developed by Galil in [11] under the assumption that the alphabet is of constant size. The notion of the *WITNESS* array was

introduced by Vishkin in [23] and was used there to design the optimal $O(\log m)$ time parallel string-matching algorithm for arbitrary alphabets. Efficient algorithms for determining approximate string matching by allowing a number of mismatches or differences have appeared in [12, 17, 18, 22]. See [13] for a survey of some of the related results. Exercise 7.34 is taken from [18].

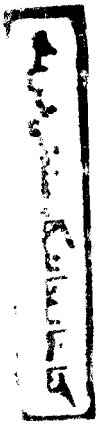
Efficient sequential algorithms for constructing suffix trees were introduced in [21, 25]. The corresponding time bound is $O(n \log \sigma)$, where n is the length of the input string and σ is the size of the alphabet. The parallel algorithm presented in the text is taken from [2]. Additional properties of strings and related algorithmic techniques are covered in [3].

Fibonacci strings (Exercise 7.5) were studied in [16]. Some related properties and references can be found there. An optimal $O(\log \log m)$ time string-matching algorithm (Exercises 7.9 and 7.18) was described in [7]. A matching lower bound has appeared in [8]. Exercise 7.12 is taken from [9]. The notions of a deterministic sample and of its application to string matching (Exercises 7.19, 7.20, and 7.21) were introduced by Vishkin [24]. Exercise 7.22 is taken from [14], whereas the parallel complexity of the string-editing problem described in Exercise 7.23 was considered in [1]. The use of hashing to reduce the space requirement of several algorithms (as in Exercise 7.25) is described in [20]. Efficient parallel algorithms can be found there. The suffix-prefix-matching problem (Exercises 7.31 and 7.33) was studied in [15]. The reduction of the two-dimensional array-matching problem to the multipattern-string matching problem (Exercise 7.35) was reported in [4, 5].

References

1. Apostolico, A., M. Atallah, L. Larmore, and H. McFaddin. Efficient parallel algorithms for string editing and related problems. Technical Report CSD-TR-724, Computer Sciences Department, Purdue University, West Lafayette, IN, 1988.
2. Apostolico, A., C. Iliopoulos, G. M. Landau, B. Schieber, and U. Vishkin. Parallel construction of a suffix tree with applications. *Algorithmica*, 3(3):347–365, 1988.
3. Apostolico, A., and Z. Galil, editors. *Combinatorial Algorithms on Words.* Springer-Verlag, New York, 1985.
4. Baker, T. P. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM J. Computing*, 7(4):533–541, 1978.
5. Bird, R. S. Two dimensional pattern matching. *Information Processing Letters*, 6(5, 10):168–170, 1977.
6. Boyer, R. S., and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20:762–772, 1977.
7. Breslauer, D., and Z. Galil. An optimal $O(\log \log n)$ time parallel string matching algorithm. *SIAM J. Computing*, 19(6):1051–1058, 1990.
8. Breslauer, D., and Z. Galil. A lower bound for parallel string matching. In *Proceedings Twentieth Annual ACM Symposium on Theory of Computing*, New Orleans, LA, 1991, pp. 439–443.
9. Eilam-Tzoreff, T., and U. Vishkin. Matching patterns in a string subject to multilinear transformations. *Theoretical Computer Science*, 60(3):231–254, 1988.
10. Fine, N. J., and H. S. Wilf. Uniqueness theorems for periodic functions. *Proc. Amer. Math. Soc.*, 16(1):109–114, 1965.

11. Galil, Z. Optimal parallel algorithms for string matching. *Information and Control*, 67(1-3):144-157, 1985.
12. Galil, Z., and R. Giancarlo. Parallel string matching with k mismatches. *Theoretical Computer Science*, 51(3):343-348, 1987.
13. Galil, Z., and R. Giancarlo. Data structures and algorithms for approximate string matching. *Journal of Complexity*, 4(1):33-72, 1988.
14. Gibbons, A., and W. Rytter. *Efficient Parallel Algorithms*, Cambridge University Press, Cambridge, Great Britain, 1988.
15. Kedem, Z., G. M. Landau, and K. Palem. Optimal parallel suffix-prefix matching algorithm and applications. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, Santa Fe, NM, 1989, pp. 388-398.
16. Knuth, D. E., J. H. Morris, and V. B. Pratt. Fast pattern matching in strings. *SIAM J. Computing*, 6(2):189-195, 1977.
17. Landau, G. M., and U. Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43(2, 3):239-249, 1986.
18. Landau, G. M., and U. Vishkin. Efficient parallel and serial approximate string matching. *Journal of Algorithms*, 10(2):157-169, 1989.
19. Lyndon, R. C., and M. P. Schutzenberger. The equation $a^m = b^n c^p$ in a free group. *Michigan Math. J.*, 9(4):289-298, 1962.
20. Matias, Y., and U. Vishkin, On parallel hashing and integer sorting. In *Proceedings of the ICALP*, Warwick, England, 1990, pp. 729-743.
21. McCreight, E. M. A space economical suffix tree construction algorithm. *JACM*, 23(2):262-272, 1976.
22. Ukkonen, E. Finding approximate patterns in strings. *Journal of Algorithms*, 6(1):132-137, 1985.
23. Vishkin, U. Optimal parallel matching in strings. *Information and Control*, 67(1-3):91-113, 1985.
24. Vishkin, U. Deterministic sampling—a new technique for fast pattern matching. *SIAM J. Computing*, 20(1):22-40, 1991.
25. Wiener, P. Linear pattern matching algorithms. In *Proceedings Fourteenth Annual Symposium on Switching and Automata Theory*, Iowa City, IA, 1973, pp. 1-11.



8

Arithmetic Computations

Many numeric problems require an excessive number of arithmetic operations, a fact that provided an initial impetus for researchers to design and build parallel computers. In particular, computations involving matrices are of utmost importance due to their many uses in solving a wide variety of applied problems. In this chapter, we shall consider a number of polynomial and matrix computations, and shall present fast parallel algorithms to handle these computations. Unfortunately, some of the described algorithms use considerably more operations than do the corresponding best-known sequential algorithms, but the underlying techniques are important in their own right and can be used to solve many related problems.

We shall start by considering computations that can be expressed as *linear recurrences*, such as polynomial evaluation and banded triangular linear systems. Section 8.1 presents a logarithmic-time algorithm for solving first-order linear recurrences using a linear number of operations. Fast parallel algorithms for *inverting triangular matrices* and for solving banded triangular linear systems are described in Section 8.2. An optimal parallel implementation of the *fast Fourier transform* (*FFT*) in logarithmic time is presented in Section 8.3. Using the FFT algorithm, we develop fast parallel algorithms for *polynomial multiplication* and *convolution* (Section 8.4), *Toeplitz matrix computations* (Section 8.5), *polynomial division* (Section 8.6), and *polynomial*

evaluation and *interpolation* (Section 8.7). The amount of work required by each of these algorithms is asymptotically the same as that required by the corresponding best-known sequential algorithms. Fast parallel algorithms for *general dense matrices* are described in Section 8.8, followed in Section 8.9 by a presentation of specialized parallel algorithms for *structured matrices* and their application to the problem of computing the *greatest common divisor.*

We shall assume the **arithmetic** PRAM model, which allows "scalar" operations on elements drawn from an arbitrary ring or field as primitive operations. This chapter does not take into consideration issues related to bit operations or to finite-precision arithmetic.

8.1 Linear Recurrences

Many arithmetic computations can be expressed iteratively such that the value y_i generated at the end of the ith iteration is a *linear combination* of the values $y_{i-1}, y_{i-2}, \ldots, y_{i-m}$ generated in the previous m iterations. Such computations, defined by a **linear recurrence of order m,** possess a straightforward efficient sequential implementation. We focus our attention in this section on the parallel implementation of first-order linear recurrences. We begin with a few examples of linear recurrences.

8.1.1 EXAMPLES OF LINEAR RECURRENCES

The problems of evaluating a polynomial at a point, determining the **LDU** factorization of a tridiagonal matrix, and solving banded triangular linear systems give rise to linear recurrences as we show next.

Polynomial Evaluation. Let $A = (a_0, a_1, \ldots, a_n)$ be an array representing the coefficients of the polynomial $p(x) = a_0 x^n + a_1 x^{n-1} + \cdots + a_{n-1}x + a_n$. We wish to compute $p(x_0)$, where x_0 is a given point.

Horner's algorithm is the classic method for computing $p(x_0)$ using n multiplications and n additions. This method is based on rewriting the expression of $p(x_0)$ as follows:

$$p(x_0) = (\cdots((a_0x_0 + a_1)x_0 + a_2)x_0 + \cdots + a_{n-1})x_0 + a_n.$$

Hence, we can obtain $p(x_0)$ by computing the innermost term $y_1 = a_0x_0 + a_1$, then the next innermost term $y_2 = y_1x_0 + a_2$, and so on, until we get $p(x_0) =$

$y_n = y_{n-1}x_0 + a_n$. This method can be expressed by the following linear recurrence:

$$y_0 = a_0$$
$$y_i = y_{i-1}x_0 + a_i, 1 \le i \le n.$$

This recurrence is a **first-order** linear recurrence since y_i depends on only y_{i-1}.

LDU Factorization of a Tridiagonal Matrix. Consider the $n \times n$ nonsingular tridiagonal matrix A given by

$$\mathbf{A} = \begin{bmatrix} b_1 & c_1 & & & & & \\ a_2 & b_2 & c_2 & & & & \\ & a_3 & b_3 & c_3 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{bmatrix}.$$

We seek the **LDU** factorization of **A**—that is, the determination of a unit lower triangular matrix **L**, a diagonal matrix **D**, and a unit upper triangular matrix **U** such that **A** = **LDU**. It is simple to verify that the matrices **L**, **D**, and **U** can be specified as follows:

$$\mathbf{L} = \begin{bmatrix} 1 & & & & & \\ l_2 & 1 & & & & \\ & l_3 & 1 & & & \\ & & \ddots & \ddots & & \\ & & & l_n & 1 \end{bmatrix}$$

$$\mathbf{U} = \begin{bmatrix} 1 & u_1 & & & & \\ & 1 & u_2 & & & \\ & & 1 & u_3 & & \\ & & & \ddots & \ddots & \\ & & & & 1 & u_{n-1} \\ & & & & & 1 \end{bmatrix}$$

$$\mathbf{D} = \begin{bmatrix} d_1 & & & & \\ & d_2 & & & \\ & & \ddots & & \\ & & & & d_n \end{bmatrix},$$

where

$$
\begin{aligned}
d_1 &= b_1 \\
d_j &= b_j - a_j c_{j-1}/d_{j-1}, 2 \le j \le n \\
l_j &= a_j/d_{j-1}, 2 \le j \le n \\
u_j &= c_j/d_j, 1 \le j \le n-1.
\end{aligned}
$$

Clearly, the values of the terms l_j and u_j can be deduced immediately from the values of d_j.

We let $d_j = w_j/w_{j-1}$, where $w_0 = 1$ and $w_1 = b_1$. Substituting $d_j = w_j/w_{j-1}$ and $d_{j-1} = w_{j-1}/w_{j-2}$ in the equation $d_j = b_j - a_j c_{j-1}/d_{j-1}$, we get $w_j = b_j w_{j-1} - (a_j c_{j-1}) w_{j-2}$. Therefore, the sequence w_j is given by the following second-order linear recurrence:

$$
\begin{aligned}
w_0 &= 1 \\
w_1 &= b_1 \\
w_j &= b_j w_{j-1} - (a_j c_{j-1}) w_{j-2}, 2 \le j \le n.
\end{aligned}
$$

It follows that the **LDU** factorization of the tridiagonal matrix **A** reduces to solving a linear recurrence of order 2.

Banded Triangular Linear Systems. Let **A** be an $n \times n$ lower triangular matrix such that all the nonzero entries of **A** lie on the main diagonal or on the $m - 1$ diagonals below it, for some integer $m < n$. For example, the matrix **A** has the following form when $m = 3$:

$$
\mathbf{A} = \begin{bmatrix}
a_{11} & 0 & 0 & 0 & \cdots & \cdots & 0 \\
a_{21} & a_{22} & 0 & 0 & \cdots & \cdots & 0 \\
a_{31} & a_{32} & a_{33} & 0 & & & \vdots \\
0 & a_{42} & a_{43} & a_{44} & \ddots & & \\
 & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\
 & & \ddots & \ddots & \ddots & \ddots & 0 \\
 & & & 0 & a_{n,n-2} & a_{n,n-1} & a_{n,n}
\end{bmatrix}.
$$

The solution to the linear system $\mathbf{Ax} = \mathbf{b}$ can be expressed by the following recurrence:

$$
\begin{aligned}
x_1 &= \frac{b_1}{a_{11}} \\
x_i &= \sum_{j=i-m+1}^{i-1} -\frac{a_{ij}}{a_{ii}} x_j + \frac{b_i}{a_{ii}}, 2 \le i \le n.
\end{aligned}
$$

We are assuming that all the entries whose indices fall outside the range $[1, 2, \ldots, n]$ are equal to 0. Therefore, the solution of the banded triangular system $\mathbf{Ax} = \mathbf{b}$ defines a linear recurrence of order $m - 1$.

8.1.2 FIRST-ORDER LINEAR RECURRENCES

We begin by developing a simple optimal parallel algorithm to compute all the terms y_i defined by the **first-order** linear recurrence:

$$\begin{aligned} y_1 &= b_1 \\ y_i &= a_i y_{i-1} + b_i, 2 \leq i \leq n. \end{aligned} \tag{8.1}$$

At first glance, the iterative nature of the computation defined by Eq. 8.1 seems to indicate a limited degree of parallelism. A moment's reflection reveals that the whole computation is similar to the prefix-sums computation (Algorithm 2.1) studied in Section 2.1. In fact, if $a_i = 1$ for all i, then the y_i expressions are the prefix sums of the elements $b_1, b_2, \ldots, b_n$. Now we will see how we can extend the prefix-sums algorithm to compute all the y_i terms defined by a first-order linear recurrence.

Let $n = 2^k$ and let the index i be even such that $2 \leq i \leq n$. Then, $y_i = a_i(a_{i-1}y_{i-2} + b_{i-1}) + b_i$, and hence $y_i = a_i a_{i-1} y_{i-2} + a_i b_{i-1} + b_i$. The resulting equations for even indices define a first-order linear recurrence of size $n/2$. More precisely, let $a'_i = a_{2i}a_{2i-1}$ and $b'_i = a_{2i}b_{2i-1} + b_{2i}$, for $1 \leq i \leq \frac{n}{2}$, and let $z_i = y_{2i}$. Then, the z_i terms satisfy the following recurrence:

$$\begin{aligned} z_1 &= b'_1 \\ z_i &= a'_i z_{i-1} + b'_i, 2 \leq i \leq n/2. \end{aligned}$$

We can now compute all the z_i values recursively. Once these values are obtained, we immediately have all the y_i values, for even indices i. If i is an odd index larger than 1, we can determine the y_i value by using the definition $y_i = a_i y_{i-1} + b_i$. The formal statement of this algorithm follows.

ALGORITHM 8.1
(First-Order Linear Recurrence)

Input: *Two arrays* $B = (b_1, b_2, \ldots, b_n)$ *and* $A = (a_1 = 0, a_2, \ldots, a_n)$ *representing the first-order linear recurrence* $y_1 = b_1$ *and* $y_i = a_i y_{i-1} + b_i$, $(2 \leq i \leq n)$; *n is assumed to be a power of* 2.
Output: *The values of all the* y_i *terms.*

begin

1. **if** $n = 1$ **then** {*set* $y_1 := b_1$, **exit**}
2. **for** $1 \leq i \leq \frac{n}{2}$ **pardo**
   Set $a'_i := a_{2i}a_{2i-1}$
   Set $b'_i := a_{2i}b_{2i-1} + b_{2i}$
3. *Recursively, solve the first-order linear recurrence defined by* $z_1 = b'_1$ *and* $z_i = a'_i z_{i-1} + b'_i$, *where* $2 \leq i \leq n/2$.

4. for $1 \leq i \leq n$ **pardo**
   i *even* : *Set* y_i: $= z_{i/2}$
   $i = 1$: *Set* y_1: $= b_1$
   i *odd* > 1 : *Set* y_i: $= a_i z_{(i-1)/2} + b_i$

end

EXAMPLE 8.1:

Let $n = 8$ in the linear recurrence defined by Eq. 8.1. During the execution of step 2 of Algorithm 8.1, we compute $a'_1 = a_2a_1 = 0, b'_1 = a_2b_1 + b_2, a'_2 = a_4a_3, b'_2 = a_4b_3 + b_4, a'_3 = a_6a_5, b'_3 = a_6b_5 + b_6, a'_4 = a_8a_7$, and $b'_4 = a_8b_7 + b_8$. The algorithm is then called recursively to compute $z_1 = y_2, z_2 = y_4, z_3 = y_6$ and $z_4 = y_8$. Executing step 2 again, we obtain $a''_1 = a'_2a'_1 = 0, b''_1 = a'_2b'_1 + b'_2, a''_2 = a'_4a'_3$ and $b''_2 = a'_4b'_3 + b'_4$. The algorithm is again called recursively. Hence, we get $a''_1 = a''_1a''_2 = 0$ and $b''_1 = a''_2b''_1 + b''_2$. We now have the value $y_8 = b''_1$. You are encouraged to verify that b''_1 is indeed the correct value of y_8. The process is now reversed. The value of y_2 is then computed, followed by the values of y_2 and y_6. Finally, we determine the values of y_1, y_3, y_5 and y_7 by using the definition of the linear recurrence. □

Theorem 8.1: *Algorithm 8.1 correctly computes the values of all the variables* y_i, *where* $1 \leq i \leq n$. *This algorithm can be implemented to run in* $O(\log n)$ *time, using a total of* $O(n)$ *arithmetic operations.*

Proof: The correctness proof follows from the discussion preceding the statement of the algorithm. As for the complexity bounds, they can be estimated as follows.

Let $T(n)$ and $W(n)$ be the running time and the total number of operations required by Algorithm 8.1. Step 1 requires $O(1)$ sequential time. Step 2 can be performed in $O(1)$ time, using a total of $O(n)$ operations; step 3 is a recursive call that takes $T(n/2)$ time and uses $W(n/2)$ operations. Finally, step 4 can be executed in $O(1)$ time, using a total of $O(n)$ operations. Therefore, $T(n)$ and $W(n)$ satisfy the following recurrences:

$$T(n) = T(n/2) + O(1)$$
$$W(n) = W(n/2) + O(n).$$

Hence, $T(n) = O(\log n)$ and $W(n) = O(n)$, as stated in the theorem. □

To account for polynomial evaluation (Section 8.1.1), we immediately obtain the following corollary to Theorem 8.1.

Corollary 8.1: *Using Horner's algorithm (Section 8.1.1) to evaluate a polynomial of degree n at a single point can be implemented to run in* $O(\log n)$ *time, using a total of* $O(n)$ *operations.*

PRAM Model: It is easy to check that no simultaneous memory access is required to implement Algorithm 8.1 within the stated bounds. Therefore, this algorithm runs on the arithmetic EREW PRAM. □

8.1.3 HIGHER-DIMENSIONAL LINEAR RECURRENCES

The scheme developed in Section 8.1.2 can be readily generalized to include first-order **higher-dimensional** linear recurrences.

Let $\{\boldsymbol{b}_i \mid 1 \leq i \leq n\}$ be a set of m-dimensional vectors, and let $\boldsymbol{A}_i$ be an $m \times m$ matrix, where $2 \leq i \leq n$. Consider the linear recurrence defined by

$$\begin{aligned} \boldsymbol{y}_1 &= \boldsymbol{b}_1 \\ \boldsymbol{y}_i &= \boldsymbol{A}_i \boldsymbol{y}_{i-1} + \boldsymbol{b}_i, \; 2 \leq i \leq n. \end{aligned} \tag{8.2}$$

As before, we reduce the problem to solving a linear recurrence defined on $\boldsymbol{y}_2, \boldsymbol{y}_4, \ldots \boldsymbol{y}_n$ by computing $\boldsymbol{A}_i' = \boldsymbol{A}_{2i}\boldsymbol{A}_{2i-1}$, and $\boldsymbol{b}_i' = \boldsymbol{A}_{2i}\boldsymbol{b}_{2i-1} + \boldsymbol{b}_{2i}$, for $1 \leq i \leq n/2$. The vector $\boldsymbol{y}_{2i}$ is then defined by $\boldsymbol{y}_{2i} = \boldsymbol{A}_i'\boldsymbol{y}_{2(i-1)} + \boldsymbol{b}_i'$. Once the vectors $\boldsymbol{y}_2, \boldsymbol{y}_4, \ldots \boldsymbol{y}_n$ have been determined, we can compute $\boldsymbol{y}_3, \boldsymbol{y}_5, \ldots \boldsymbol{y}_{n-1}$ directly from the definition of the linear recurrence, that is, $\boldsymbol{y}_3 = \boldsymbol{A}_3\boldsymbol{y}_2 + \boldsymbol{b}_3$, $\boldsymbol{y}_5 = \boldsymbol{A}_5\boldsymbol{y}_4 + \boldsymbol{b}_5$, and so on. In fact, the algorithm is identical to Algorithm 8.1, except that it uses matrices and vectors instead of scalars.

Before analyzing the complexity bounds of this scheme, let us recall from Section 5.5 that two $m \times m$ matrices can be multiplied in $O(\log m)$ time, using $O(M(m))$ operations, where $M(m)$ is the number of arithmetic operations required to compute the product of two $m \times m$ matrices sequentially. The current best known bound is $M(m) = O(m^{2.376})$ over a ring.

Theorem 8.2: *The first-order linear recurrence defined by Eq. 8.2, where the matrices $\boldsymbol{A}_i$ are each of dimension $m \times m$, and the vectors $\boldsymbol{y}_i$ and $\boldsymbol{b}_i$ are each of dimension m, can be computed in $O(\log n \log m)$ time, using a total of $O(nM(m))$ arithmetic operations, where $M(m)$ is the best known sequential bound for multiplying two $m \times m$ matrices.*

Proof: The correctness proof follows by a simple induction on n.

Let $T(n, m)$ and $W(n, m)$ be the running time and the total number of operations required by the scheme described before the statement of the theorem. The algorithm consists of three main steps, each of which is analyzed separately.

- *Step 1*: This step involves the computation of $\boldsymbol{A}_i' = \boldsymbol{A}_{2i}\boldsymbol{A}_{2i-1}$, and $\boldsymbol{b}_i' = \boldsymbol{A}_{2i}\boldsymbol{b}_{2i-1} + \boldsymbol{b}_{2i}$, for $1 \leq i \leq n/2$. The matrices $\boldsymbol{A}_i'$ can be computed in $O(\log m)$ time, using a total of $O(nM(m))$ operations, since the matrices involved are each of size $m \times m$; the vectors $\boldsymbol{b}_i'$ can be computed in $O(\log m)$ time, using $O(nm^2) = O(nM(m))$ operations.

- *Step 2*: This step makes a recursive call to solve the linear recurrence defined by $\boldsymbol{y}_{2i} = \boldsymbol{A}_i'\boldsymbol{y}_{2(i-1)} + \boldsymbol{b}_i'$ which requires $T(n/2, m)$ time, using a total of $W(n/2, m)$ operations.
- *Step 3*: This step computes $\boldsymbol{y}_{2i+1} = \boldsymbol{A}_{2i+1}\boldsymbol{y}_{2i} + \boldsymbol{b}_{2i+1}$, for each i such that $1 \leq i \leq (n/2) - 1$. Hence, this step can be performed in $O(\log m)$ time, using $O(nm^2) = O(nM(m))$ operations.

Therefore, $T(n, m)$ and $W(n, m)$ satisfy the following recurrences:

$$T(n, m) = T(n/2, m) + O(\log m)$$
$$W(n, m) = W(n/2, m) + O(nM(m)).$$

Since the solutions of these recurrences are $T(n, m) = O(\log n \log m)$ and $W(n, m) = O(nM(m))$, the theorem follows. □

In many important cases, we can reduce the required number of operations, as will be shown in the next two examples.

EXAMPLE 8.2:

Suppose that all the matrices $\boldsymbol{A}_i$ of Eq. 8.2 are equal to $\boldsymbol{A}$ of size $n \times n$ (and hence $m = n$). In this case, $\boldsymbol{y}_i = \boldsymbol{A}^{i-1}\boldsymbol{b}_1 + \boldsymbol{A}^{i-2}\boldsymbol{b}_2 + \cdots + \boldsymbol{A}\boldsymbol{b}_{i-1} + \boldsymbol{b}_i$, for $1 \leq i \leq n$. During the first iteration, we compute $\boldsymbol{A}_i' = \boldsymbol{A}_{2i}\boldsymbol{A}_{2i-1} = \boldsymbol{A}^2$, and $\boldsymbol{b}_i' = \boldsymbol{A}_{2i-1}\boldsymbol{b}_{2i-1} + \boldsymbol{b}_{2i} = \boldsymbol{A}\boldsymbol{b}_{2i-1} + \boldsymbol{b}_{2i}$, for $1 \leq i \leq n/2$. Computation of the $\boldsymbol{b}_i'$ vectors can be expressed in the following matrix form (where n is assumed to be a power of 2):

$$[\boldsymbol{b}_1', \boldsymbol{b}_2', \ldots, \boldsymbol{b}_{\frac{n}{2}}'] = \boldsymbol{A}[\boldsymbol{b}_1, \boldsymbol{b}_3, \ldots, \boldsymbol{b}_{n-1}] + [\boldsymbol{b}_2, \boldsymbol{b}_4, \ldots, \boldsymbol{b}_n].$$

It follows that this iteration can be done in $O(\log n)$ time, using $O(M(n))$ operations.

We can obtain the complete solution (step 4 of Algorithm 8.1) by using the same scheme sketched for computing the $\boldsymbol{b}_i'$ vectors. Therefore, the overall running time is $O(\log^2 n)$, and the total number of arithmetic operations is $O(M(n) \log n)$. □

EXAMPLE 8.3: (Krylov Matrix)

Let $\boldsymbol{v}$ be an arbitrary m-dimensional vector. Suppose that $\boldsymbol{b}_1 = \boldsymbol{v}, \boldsymbol{b}_i = 0$ for $2 \leq i \leq n$, and $\boldsymbol{A}_i = \boldsymbol{A}$ in Eq. 8.2. Clearly, in this case, $\boldsymbol{y}_i = \boldsymbol{A}^{i-1}\boldsymbol{v}$, for $1 \leq i \leq n$. The matrix $[\boldsymbol{v}, \boldsymbol{A}\boldsymbol{v}, \boldsymbol{A}^2\boldsymbol{v}, \cdots, \boldsymbol{A}^{n-1}\boldsymbol{v}]$ is called the *Krylov matrix* defined by the matrix $\boldsymbol{A}$, the vector $\boldsymbol{v}$, and the integer n. This is a special case of the linear recurrence introduced in Example 8.2. Using the same scheme, we obtain an algorithm to compute the Krylov matrix in $O(\log n \log m)$ time, using a total of $O(M(m) \log n)$ operations, assuming $n \leq m$. The importance of the Krylov matrix will be revealed in Section 8.8. □

Having considered first-order linear recurrences, we turn now to parallel algorithms for higher-order linear recurrences. We start with triangular linear systems.

8.2 Triangular Linear Systems

Solving a linear system of equations is one of the most fundamental problems in numeric computations. The classic Gaussian elimination scheme to solve an arbitrary linear system of equations reduces the given system to a triangular form and then generates the solution by using the standard forward substitution algorithm. Our goal here is to present a simple, fast parallel algorithm for solving triangular linear systems. The general case will be handled in Section 8.8.

Consider the linear system of equations $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{x}$ and $\mathbf{b}$ are n-dimensional vectors, and $\mathbf{A}$ is the following $n \times n$ unit lower triangular matrix:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ a_{21} & 1 & 0 & \cdots & 0 \\ a_{31} & a_{32} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{n,n-1} & 1 \end{bmatrix}.$$

For a given arbitrary nonsingular lower triangular system, we can make all the entries on the diagonal equal to 1 by dividing the ith equation by a_{ii}, whenever $a_{ii} \neq 1$, for $1 \leq i \leq n$. Hence, our assumption of a unit diagonal is not a restriction.

The simplest method to evaluate $\mathbf{x}$ is perhaps to compute successively $x_1 = b_1, x_2 = b_2 - a_{21}x_1, x_3 = b_3 - a_{31}x_1 - a_{32}x_2, \cdots, x_i = b_i - \sum_{j=1}^{i-1} a_{ij}x_j, \ldots$. This scheme clearly requires $O(n^2)$ arithmetic operations. Unfortunately, such a simple method does not seem to lead to a fast parallel algorithm for computing all the x_i's. An alternative method exists that can be implemented in $O(\log^2 n)$ time, but it requires $O(M(n))$ arithmetic operations.

8.2.1 A FAST PARALLEL MATRIX-INVERSION ALGORITHM

We shall use a simple divide-and-conquer strategy to invert the $n \times n$ unit lower triangular matrix $\mathbf{A}$. Once $\mathbf{A}^{-1}$ is generated, determining $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ can be done in $O(\log n)$ time, using a total of $O(n^2)$ operations.

Let **A** be partitioned into $(n/2) \times (n/2)$ blocks as follows:

$$A = \begin{bmatrix} A_1 & 0 \\ A_2 & A_3 \end{bmatrix},$$

where n is assumed to be a power of 2. Clearly, A_1 and A_3 are nonsingular, lower triangular matrices. Using straightforward matrix multiplication, it is easy to verify that $AA^{-1} = I_n$, where I_n is the $n \times n$ identity matrix and $\mathbf{A}^{-1}$ is given by

$$A^{-1} = \begin{bmatrix} A_1^{-1} & 0 \\ -A_3^{-1}A_2A_1^{-1} & A_3^{-1} \end{bmatrix}.$$

Therefore, we can obtain the inverse of A by recursively computing the inverses of A_1 and A_3, and by performing two $(n/2) \times (n/2)$ matrix multiplications to generate the term $-A_3^{-1}A_2A_1^{-1}$. This divide-and-conquer method leads to the following theorem.

Theorem 8.3: *An $n \times n$ triangular linear system of equations can be solved in $O(\log^2 n)$ time, using $O(M(n))$ arithmetic operations.*

Proof: The time bound of $O(\log^2 n)$ follows from the observation that the algorithm runs for $O(\log n)$ iterations, each of which involves two matrix multiplications of size $\leq n/2$. As for the total number of arithmetic operations, they can be estimated as follows.

Let $W(n)$ be the number of arithmetic operations needed to invert an $n \times n$ lower triangular matrix. Then, $W(n)$ satisfies the following recurrence:

$$W(n) = 2W(n/2) + 2M(n/2).$$

We can check that $W(n) \leq M(n)$ satisfies this recurrence, if we make the reasonable assumption that $M(n)$ satisfies the relation $M(n) \geq 4M(n/2)$. □

PRAM Model: Matrix multiplication is the basic routine used to obtain our desired solution. Hence, we need the arithmetic CREW PRAM model to implement this algorithm within the stated bounds. □

Exercises 8.5 and 8.6 provide two other fast algorithms to invert lower triangular matrices.

8.2.2 BANDED TRIANGULAR LINEAR SYSTEMS

One important special class of triangular systems arises in numerous applications. This class comprises the **banded** triangular linear systems, in which the coefficient matrix A has a small number of subdiagonals (diagonals below

the main diagonal) with nonzero entries. It turns out that a combination of the scheme for solving first-order linear recurrences and the fast scheme for solving triangular linear systems yields a fast and efficient parallel algorithm for solving banded triangular linear systems.

A lower triangular matrix A is of **bandwidth** m if $a_{ij} = 0$ whenever $i - j \geq m$ or $i < j$, and hence all the nonzero entries of A belong to the main diagonal and to the $m - 1$ subdiagonals below it.

EXAMPLE 8.4:

Consider the case of a banded triangular system in which $n = 6$ and $m = 3$. Then, the matrix **A** has the form

$$\begin{bmatrix} a_{11} & 0 & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & 0 \\ 0 & a_{42} & a_{43} & a_{44} & 0 & 0 \\ 0 & 0 & a_{53} & a_{54} & a_{55} & 0 \\ 0 & 0 & 0 & a_{64} & a_{65} & a_{66} \end{bmatrix}.$$

□

Without loss of generality, we can assume that the elements $\{a_{ii}\}$ on the main diagonal are all equal to 1. The solution to the system $\mathbf{Ax} = \mathbf{b}$ can then be expressed by the following recurrence:

$$x_1 = b_1$$

$$x_i = \sum_{j=i-m+1}^{i-1} - a_{ij}x_j + b_i, 2 \leq i \leq n.$$

All entries whose indices fall outside the range $[1, 2, \ldots, n]$ are assumed to be zeros. Hence a banded triangular linear system of bandwidth m gives rise to an $(m - 1)$st-order linear recurrence.

We can use the fast method developed in Section 8.2.1 for solving arbitrary triangular linear systems. The corresponding algorithm would be highly inefficient, since the straightforward sequential algorithm implied by the recurrence requires only $O(mn)$ operations, which is linear in n when m is a constant. An alternative method that depends both on the algorithm for solving triangular systems and on the algorithm for the parallel evaluation of first-order linear recurrences will be developed next.

Consider the banded triangular linear system $\boldsymbol{Ax} = \boldsymbol{b}$, where A is an $n \times n$ matrix of bandwidth m. Assume that m divides n and partition **A** into $m \times m$ blocks $A_{i,j}$, where $1 \leq i, j \leq n/m$. The resulting matrix is block-bidiagonal such that the blocks $A_{i,i}$ on the main diagonal are lower triangular and the blocks $A_{i,i-1}$ on the subdiagonal are upper triangular.

EXAMPLE 8.5:

Let $n = 9$ and $m = 3$. Then A is partitioned as follows:

$$A = \left[\begin{array}{ccc|ccc|ccc} a_{11} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & a_{42} & a_{43} & a_{44} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & a_{53} & a_{54} & a_{55} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & a_{64} & a_{65} & a_{66} & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & a_{75} & a_{76} & a_{77} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & a_{86} & a_{87} & a_{88} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & a_{97} & a_{98} & a_{99} \end{array}\right].$$

□

We let D be the block diagonal matrix consisting of the submatrices $A_{1,1}$, $A_{2,2}, \cdots, A_{\frac{n}{m},\frac{n}{m}}$; that is,

$$D = \begin{bmatrix} A_{1,1} & & & \\ & A_{2,2} & & \\ & & \ddots & \\ & & & A_{\frac{n}{m},\frac{n}{m}} \end{bmatrix}.$$

We let $A^* = D^{-1}A$, and let $d = D^{-1}b$. Notice that A^* is block-bidiagonal such that the main diagonal consists of n/m blocks of I_m, and the subdiagonal consists of blocks $A^*_{i,i-1} = A_{i,i}^{-1}A_{i,i-1}$, for $2 \leq i \leq n/m$. The computation of $\mathbf{A^*}$ requires $(n/m) - 1$ independent $m \times m$ lower triangular matrix inversions for generating $\{A_{i,i}^{-1}\}_{i=2}^{n/m}$, and $(n/m) - 1$ independent $m \times m$ matrix multiplications for generating $\{A_{i,i}^{-1}A_{i,i-1}\}_{i=2}^{n/m}$. Using our algorithm for inverting lower triangular matrices, we can compute the blocks $A_{i,i}^{-1}$, in $O(\log^2 m)$ time, using a total of $O((n/m)M(m))$ operations, for all $1 \leq i \leq n/m$. The $(n/m) - 1$ matrix multiplications clearly can be computed within these bounds. The computation of the vector $d = D^{-1}b$ can also be computed within these bounds.

It is clear that the solution of the linear system $Ax = b$ is also given by $D^{-1}Ax = D^{-1}b$, that is, $A^*x = d$, where A^* has the form

$$\begin{bmatrix} I_m & & & & \\ A^*_{2,1} & I_m & & & \\ & A^*_{3,2} & I_m & & \\ & & \ddots & \ddots & \\ & & & A^*_{\frac{n}{m},\frac{n}{m}-1} & I_m \end{bmatrix}.$$

We partition $\mathbf{x}$ and $\mathbf{d}$ into the m-dimensional subvectors $x = (x_1, x_2, \ldots, x_{n/m})$ and $d = (d_1, d_2, \ldots, d_{n/m})$, respectively. The solution to the system $A^*x = d$ can now be expressed by the following first-order linear recurrence:

$$x_1 = d_1$$
$$x_i = -A^*_{i,i-1}x_{i-1} + d_i, 2 \leq i \leq n/m$$

This recurrence is precisely the type of linear recurrence we considered in Section 8.1.3. Using Theorem 8.2, all the vectors x_i's can be obtained in $O(\log (n/m) \log m)$ time, using a total of $O((n/m)M(m))$ operations. Therefore, we have the following theorem.

Theorem 8.4: *The solution of an $n \times n$ lower triangular linear system of equations of bandwidth m can be determined in $O(\log n \log m)$ time, using a total of $O((n/m)M(m))$ arithmetic operations.* □

Recall that $M(m) < m^{2.376}$ over a ring, and hence the total number of arithmetic operations required by the algorithm for solving banded triangular linear systems is $O(n.m^{1.376})$ over a ring. This method thus compares favorably with the $O(nm)$ time sequential algorithm; in particular, it is optimal in the case when m is a constant.

PRAM Model: The basic routines used in the algorithm described in this section are those for inverting lower triangular matrices, and for solving an m-dimensional first-order linear recurrence. These routines require matrix multiplication and prefix-sums computations. Therefore, our algorithm runs on the arithmetic CREW PRAM model within the stated bounds. □

8.3 The Discrete Fourier Transform

The **discrete fourier transform (DFT)** is an extremely important tool used heavily in digital signal processing. It can also serve as a basic computational tool to derive efficient algorithms for a diverse collection of problems. The wide applicability of this transform stems primarily from the existence of efficient and elegant schemes to compute the transform.

We begin our presentation by describing a **Fast Fourier Transform (FFT)** algorithm, which can be implemented in $O(\log n)$ time, using a total of $O(n \log n)$ operations, on the EREW PRAM. Several applications of the FFT algorithm—such as polynomial multiplication, convolution, polynomial division, and polynomial evaluation and interpolation—will be considered in

Sections 8.4 through 8.7. The total amount of work performed by the resulting algorithms matches the corresponding time taken by the best known sequential algorithms.

We begin by defining the discrete Fourier transform.

8.3.1 THE DISCRETE FOURIER TRANSFORM DEFINED OVER THE COMPLEX FIELD

We let $\mathbb{C}$ be the field of complex numbers, and let n be an integer. Define the complex number $\omega = e^{i\frac{2\pi}{n}} = \cos\frac{2\pi}{n} + i\sin\frac{2\pi}{n}$, where $i = \sqrt{-1}$. Clearly, $\omega^n = e^{2\pi i} = 1$, and $1, \omega, \omega^2, \ldots, \omega^{n-1}$ constitute all the (distinct) nth roots of unity in the complex field $\mathbb{C}$. The complex number ω is called a **primitive** nth root of unity.

We let W_n be the $n \times n$ matrix whose rows and columns are indexed $0, 1, \ldots, n-1$ such that $W_n(j,k) = \omega^{jk}$, where $0 \le j, k \le n-1$. The DFT of an n-dimensional vector x is defined as the vector $y = W_n x$. Therefore, $y_j = \sum_{k=0}^{n-1} \omega^{jk} x_k$, $0 \le j \le n-1$.

EXAMPLE 8.6:

Let $n = 4$ and hence $\omega = e^{i\frac{2\pi}{4}} = \cos\frac{\pi}{2} + i\sin\frac{\pi}{2} = i$. The four distinct fourth roots of unity are $i, i^2 = -1, i^3 = -i, i^4 = 1$. The matrix W_4 is given by

$$W_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}.$$

The DFT of a four-dimensional vector x is $y = W_4 x$, where the components are

$$\begin{aligned} y_0 &= x_0 + x_1 + x_2 + x_3 \\ y_1 &= x_0 + ix_1 - x_2 - ix_3 \\ y_2 &= x_0 - x_1 + x_2 - x_3 \\ y_3 &= x_0 - ix_1 - x_2 + ix_3. \end{aligned}$$

□

Since the DFT of x can be viewed as a matrix-vector product, it can be computed by using the standard algorithm for multiplying an arbitrary matrix by a vector. Such an algorithm can be implemented in $O(\log n)$ time using $\Theta(n^2)$ arithmetic operations involving complex numbers. The FFT algorithm presented next uses only $O(n \log n)$ arithmetic operations.

8.3.2 THE FAST FOURIER TRANSFORM ALGORITHM

We shall use a divide-and-conquer strategy to compute the DFT $y = W_n x$. For the remainder of this section, we assume that n is a power of 2.

Let j be an even index—say, $j = 2l$—such that $0 \le l \le \frac{n}{2} - 1$. Then, according to the definition of the DFT, we have $y_j = y_{2l} = \sum_{k=0}^{n-1} \omega^{2lk} x_k$, and hence,

$$\begin{aligned} y_{2l} = {} & x_0 + \omega^{2l} x_1 + \omega^{4l} x_2 + \ldots + \omega^{2l(\frac{n}{2}-1)} x_{\frac{n}{2}-1} + x_{\frac{n}{2}} + \\ & \omega^{2l} x_{\frac{n}{2}+1} + \omega^{4l} x_{\frac{n}{2}+2} + \ldots + \omega^{2l(\frac{n}{2}-1)} x_{n-1}, \end{aligned} \tag{8.3}$$

where we have used the fact that $\omega^{ln} = (\omega^n)^l = 1$. It follows that $y_{2l} = (x_0 + x_{\frac{n}{2}}) + \omega^{2l}(x_1 + x_{\frac{n}{2}+1}) + \omega^{4l}(x_2 + x_{\frac{n}{2}+2}) + \ldots + \omega^{2l(\frac{n}{2}-1)}(x_{\frac{n}{2}-1} + x_{n-1})$.

Clearly, $\omega^2 = e^{i\frac{2\pi}{(\frac{n}{2})}}$, and hence ω^2 is a primitive $(n/2)$th root of unity. It follows that the vector $z^{(1)} = [y_0, y_2, \ldots, y_{n-2}]^T$ is the DFT of the vector $[x_0 + x_{\frac{n}{2}}, x_1 + x_{\frac{n}{2}+1}, \ldots, x_{\frac{n}{2}-1} + x_{n-1}]^T$.

If the index j is odd—say, $j = 2l + 1$—we can show in a similar fashion that

$$\begin{aligned} y_{2l+1} = {} & (x_0 + \omega^{\frac{n}{2}} x_{\frac{n}{2}}) + \omega^{2l}(\omega x_1 + \omega^{\frac{n}{2}+1} x_{\frac{n}{2}+1}) + \ldots + \\ & \omega^{2l(\frac{n}{2}-1)}(\omega^{\frac{n}{2}-1} x_{\frac{n}{2}-1} + \end{aligned}$$

We can simplify this expression by noting that $\omega^{\frac{n}{2}} = e^{i\pi} = -1$, and thus

$$\begin{aligned} y_{2l+1} = {} & (x_0 - x_{\frac{n}{2}}) + \omega^{2l}.\omega\,(x_1 - x_{\frac{n}{2}+1}) + \\ & \omega^{4l}.\omega^2\,(x_2 - x_{\frac{n}{2}+2}) + \ldots + \\ & \omega^{2l(\frac{n}{2}-1)}.\omega^{\frac{n}{2}-1}\,(x_{\frac{n}{2}-1} - x_{n-1}). \end{aligned} \tag{8.4}$$

Therefore, the vector $z^{(2)} = [y_1, y_3, \ldots, y_{n-1}]^T$ is the DFT of the vector $[x_0 - x_{\frac{n}{2}}, \omega\,(x_1 - x_{\frac{n}{2}+1}), \ldots, \omega^{\frac{n}{2}-1}(x_{\frac{n}{2}-1} - x_{n-1})]^T$. This fact is illustrated pictorially in Fig. 8.1.

We are now ready to state our algorithm more formally.

ALGORITHM 8.2

(Fast Fourier Transform)

Input: *An n-dimensional vector* $\mathbf{x}$ *whose entries are complex numbers, and* $\omega = e^{i\frac{2\pi}{n}}$, *where n is assumed to be a power of* 2.

Output: *The vector* $\mathbf{y}$ *that is the DFT of* $\mathbf{x}$.

begin

1. **if** $n = 2$ **then** {*Set* $y_1 := x_1 + x_2, y_2 := x_1 - x_2$, **exit**}

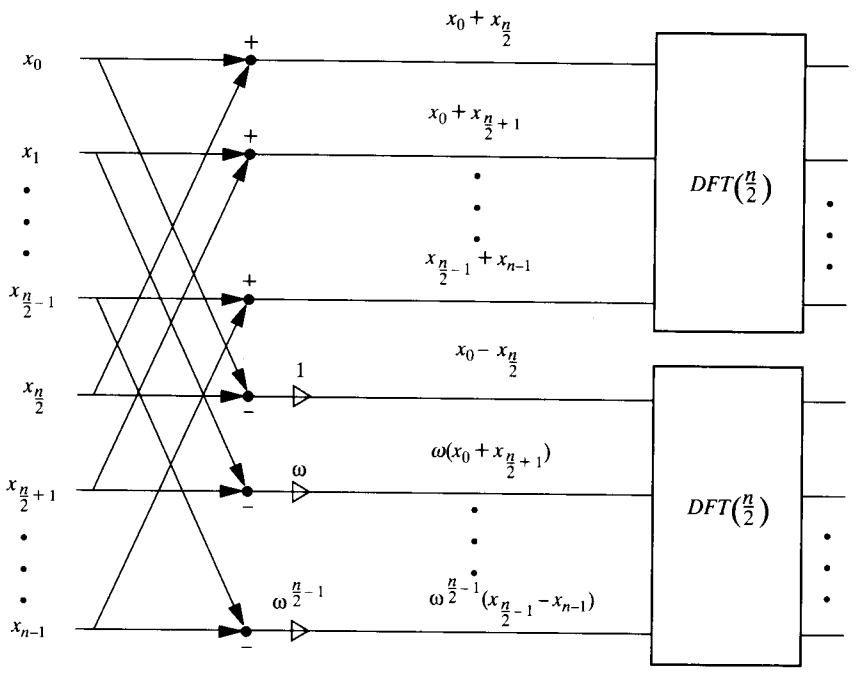


FIGURE 8.1
An illustration of the FFT algorithm. The top $DFT(\frac{n}{2})$ block generates the even-indexed entries, and the bottom $DFT(\frac{n}{2})$ block generates the odd-indexed entries.

2. **for** $0 \le l \le \frac{n}{2} - 1$ **pardo**
   Set u_l: $= x_l + x_{\frac{n}{2}+l}$
   Set v_l: $= \omega^l(x_l - x_{\frac{n}{2}+l})$
3. *Recursively, compute the DFT of the two vectors* $[u_0, u_1, \cdots, u_{\frac{n}{2}-1}]$ *and* $[v_0, v_1, \cdots, v_{\frac{n}{2}-1}]$, *and store the results in vectors* $z^{(1)} = [z_0^{(1)}, z_1^{(1)}, \cdots, z_{\frac{n}{2}-1}^{(1)}]$ *and* $z^{(2)} = [z_0^{(2)}, z_1^{(2)}, \cdots, z_{\frac{n}{2}-1}^{(2)}]$, *respectively.*
4. **for** $0 \le j \le n - 1$ **pardo**
   j even: Set y_j: $= z^{(1)}_{\frac{j}{2}}$
   j odd: Set y_j: $= z^{(2)}_{\frac{j-1}{2}}$

end

EXAMPLE 8.7:

Let us apply the FFT algorithm to compute the DFT of the vector $[2, i, 1 - i, 0, 1, -i, 0, 0]$. In this case, $\omega = e^{i\frac{2\pi}{8}} = \cos\frac{\pi}{4} + i\sin\frac{\pi}{4} = (\sqrt{2}/2) + i(\sqrt{2}/2)$, and the coefficient matrix is given by

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega & i & \omega' & -1 & -\omega & -i & -\omega' \\ 1 & i & -1 & -i & 1 & i & -1 & -i \\ 1 & \omega' & -i & \omega & -1 & -\omega' & i & -\omega \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & -\omega & i & -\omega' & -1 & \omega & -i & \omega' \\ 1 & -i & -1 & i & 1 & -i & -1 & i \\ 1 & -\omega' & -i & -\omega & -1 & \omega' & i & \omega \end{bmatrix}.$$

where $\omega' = -\frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2}$. At step 2 of Algorithm 8.2, the following quantities are computed:

$$\begin{aligned} u_0 &= x_0 + x_4 = 3 & v_0 &= x_0 - x_4 = 1 \\ u_1 &= x_1 + x_5 = 0 & v_1 &= \omega(x_1 - x_5) = 0 \\ u_2 &= x_2 + x_6 = 1 - i & v_2 &= \omega^2(x_2 - x_6) = 1 + i \\ u_3 &= x_3 + x_7 = 0 & v_3 &= \omega^3(x_3 - x_7) = 0. \end{aligned}$$

The algorithm is now called recursively to compute the DFTs of the two vectors determined by the components u_j and v_j. Let us concentrate on the vector corresponding to the elements u_j. Again, at step 2, we obtain

$$\begin{aligned} u_0' &= u_0 + u_2 = 4 - i & v_0' &= u_0 - u_2 = 2 + i \\ u_1' &= u_1 + u_3 = 0 & v_1' &= \omega^2(u_1 - u_3) = 0. \end{aligned}$$

Since the size of each of these vectors is 2, the next recursive calls return $[u_0' + u_1', u_0' - u_1'] = [4 - i, 4 - i]$ and $[v_0' + v_1', v_0' - v_1'] = [2 + i, 2 + i]$. Hence, the DFT of the vector determined by the components u_i is given by $[4 - i, 2 + i, 4 - i, 2 + i]$.

Similarly, we can go through the same process to determine the DFT of the vector whose components are the elements v_j. As a result, we obtain $[2 + i, -i, 2 + i, -i]$. Combining these two vectors as in step 4 of Algorithm 8.2, we get $y = [4 - i, 2 + i, 2 + i, -i, 4 - i, 2 + i, 2 + i, -i]^T$. □

Figure 8.2 illustrates the FFT algorithm that results from unfolding the recursion for $n = 8$.

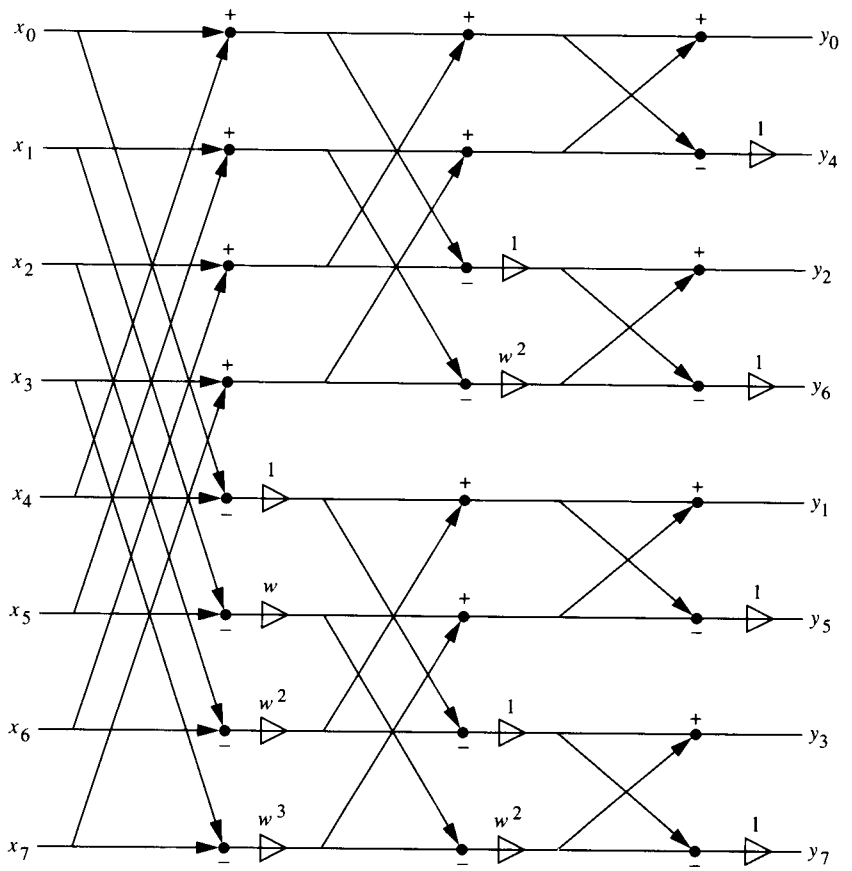


FIGURE 8.2
The FFT algorithm for $n = 8$.

Theorem 8.5: *The FFT Algorithm (Algorithm 8.2) computes the DFT of an n-dimensional vector in $O(\log n)$ time, using a total of $O(n \log n)$ arithmetic operations.*

Proof: The correctness proof follows from Eqs. 8.3 and 8.4. As for the complexity bounds, it is easy to check that the running time $T(n)$ and the total number $W(n)$ of operations satisfy the following recurrences:

$$T(n) = T(n/2) + O(1)$$

$$W(n) = 2W(n/2) + O(n),$$

and hence $T(n) = O(\log n)$ and $W(n) = O(n \log n)$. □

PRAM Model: The value of ω can be distributed to all the processors, and we can compute all the powers of ω by using a prefix-sums algorithm. At each parallel step, we know precisely the powers of ω that are needed. It follows that the FFT algorithm can be implemented on the arithmetic EREW PRAM within the stated bounds. □

It is easy to verify, by using matrix multiplication, that the inverse of the matrix W_n is given by $W_n^{-1}(j,k) = \frac{1}{n}\omega^{-jk}$, for $0 \leq j,k \leq n-1$. We define the **inverse discrete Fourier transform (IDFT)** as the vector $y = W_n^{-1}x$. The complexity bounds of computing the IDFT are stated in the following theorem.

Theorem 8.6: *Given an n-dimensional vector* x*, the IDFT of* x *can be computed in* $O(\log n)$ *time, using* $O(n \log n)$ *operations on the EREW PRAM.*

Proof: An FFT-type algorithm can be used to achieve the bounds stated in the theorem. The proof is left to Exercise 8.8. □

8.3.3 *THE DISCRETE FOURIER TRANSFORM OVER A RING

The Fourier transform has been defined over the complex field. In some applications, it is useful to extend this notion to an arbitrary commutative ring $(R, +, .)$.

An element ω of a commutative ring R is a **primitive nth root of unity** if ω generates a multiplicative group of order n. That is, the elements ω, $\omega^2, \ldots, \omega^{n-1}, \omega^n = 1$ are all distinct. In this case, we can use the usual matrix-vector formulation to define the DFT of an n-dimensional vector over R. The FFT algorithm can then be applied just as in the case of the complex numbers.

An important special case is when R is the ring Z_m of integers modulo m, for some integer m. In this case, we can specify values of m for which certain primitive roots of unity exist.

Let G_m be the set of the elements $k \in Z_m$ that are relatively prime to m. It can be shown that $(G_m,.)$ is a cyclic group of order $\phi(m)$, where $\phi(m)$ is the number of integers less than m that are relatively prime to m. Hence, every element $\alpha \in G_m$ satisfies the equation $\alpha^{\phi(m)} - 1 = 0 \bmod m$. Any generator ω of G_m is a primitive root of unity.

Expanding these preliminary facts, we can show the following theorem.

Theorem 8.7: *Given any* ω *and* n *such that* $\omega = 2^p$ *and* $n = 2^q$ *for some* p *and* q, *the ring* Z_m *has* ω *as a primitive* n*th root of unity, where* $m = \omega^{n/2} + 1$. □

The interested reader can consult the bibliographic notes at the end of this chapter for several references containing more detailed presentation of the DFT over a ring.

EXAMPLE 8.8:

Let $R = Z_9$, the ring of integers modulo 9. This ring consists of the elements $\{0, 1, 2, 3, 4, 5, 6, 7, 8\}$. The multiplicative group G_9 consists of $\{1, 2, 4, 5, 7, 8\}$. These are all the elements relatively prime to 9. Among the six elements of G_9, only 2 and 5 are primitive sixth roots of unity. For instance, $\omega = 2, \omega^2 = 4, \omega^3 = 8, \omega^4 = 7, \omega^5 = 5$, and $\omega^6 = 1$. On the other hand, for $\alpha = 4$, we have $\alpha = 4, \alpha^2 = 7$, and $\alpha^3 = 1$; thus, $\alpha = 4$ is not a primitive sixth root of unity. □

EXAMPLE 8.9:

Suppose we wish to perform an eight-dimensional DFT with $\omega = 2$ as a primitive eighth root of unity. Let $m = 2^4 + 1 = 17$, and consider the ring Z_{17}. Then, $\omega = 2, \omega^2 = 4, \omega^3 = 8, \omega^4 = 16, \omega^5 = 15, \omega^6 = 13, \omega^7 = 9$, and $\omega^8 = 1$. Hence, $\omega = 2$ is indeed a primitive eighth root of unity in Z_{17}, as stated in Theorem 8.7. □

8.4 Polynomial Multiplication and Convolution

Consider the polynomial $p(x) = \sum_{k=0}^{n-1} a_k x^k$, where the coefficients a_k are elements from a ring that possesses an nth primitive root of unity (the reader may wish to assume that the a_k's are complex numbers). So far, we have assumed that such a polynomial is represented by the list of its coefficients $(a_0, a_1, \ldots, a_{n-1})$. On the other hand, it is well known that $p(x)$ is also uniquely determined by its values $y_j = p(x_j)$ at *any* n distinct points $x_0, x_1, \ldots, x_{n-1}$.

Polynomial evaluation at n distinct points can be viewed as the process of converting the list of coefficients representation into the representation of $p(x)$ by its values at the n distinct points. The reverse process of generating the list of coefficients from the set of values (x_j, y_j), where $0 \le j \le n - 1$, is called **polynomial interpolation.** We shall now relate the DFT computation to the notion of polynomial evaluation and polynomial interpolation.

8.4.1 POLYNOMIAL EVALUATION AND INTERPOLATION AT ROOTS OF UNITY

Let $\boldsymbol{y}$ be the DFT of the vector $\boldsymbol{a}$, where $\boldsymbol{a} = [a_0, a_1, \ldots, a_{n-1}]^T$ and the elements a_i represent the coefficients of the polynomial $p(x)$. Then $y_j = \sum_{k=0}^{n-1} a_k(\omega^j)^k$, where $0 \le j \le n-1$. Hence, y_j is the value of $p(x)$ at the point ω^j for each j. Therefore, the DFT of $\boldsymbol{a}$ can be viewed as the ordered list of the values of the $(n-1)$st-degree polynomial $p(x)$ at the points $1, \omega, \omega^2, \ldots, \omega^{n-1}$. Recall that, since ω is a primitive nth root of unity, the points ω^j, where $0 \le j \le n-1$, are all distinct.

On the other hand, $\boldsymbol{y} = \boldsymbol{W}_n\boldsymbol{a}$ is equivalent to $\boldsymbol{a} = \boldsymbol{W}_n^{-1}\boldsymbol{y}$, and hence computing the inverse DFT of the vector $\boldsymbol{y}$ corresponds to generating the list of the coefficients of $p(x)$—that is, to polynomial interpolation—where the n distinct evaluation points are the n roots of unity.

Evaluation of an $(n-1)$st-degree polynomial at the distinct n roots of unity and the corresponding polynomial interpolation can each be achieved in $O(\log n)$ time, using a total of $O(n \log n)$ arithmetic operations.

Let us examine what are the implications of the relationship between DFT computation and evaluation/interpolation for polynomial multiplication.

8.4.2 POLYNOMIAL MULTIPLICATION USING THE FFT ALGORITHM

The product of two polynomials $p(x) = \sum_{k=0}^{n-1} a_k x^k$ and $q(x) = \sum_{k=0}^{m-1} b_k x^k$ is the polynomial $r(x) = p(x)q(x) = \sum_{k=0}^{n+m-2} c_k x^k$, such that $c_k = \sum_{j=0}^{k} a_j b_{k-j}$, where a_k and b_{j-k} are assumed to be 0 if their indices fall outside the range $[0, 1, \ldots, n-1]$ and $[0, 1, \ldots, m-1]$, respectively.

EXAMPLE 8.10:

Let $p(x) = a_0 + a_1x$ and $q(x) = b_0 + b_1x + b_2x^2$. Then $r(x) = (a_0 + a_1x)(b_0 + b_1x + b_2x^2) = a_0b_0 + (a_0b_1 + a_1b_0)x + (a_0b_2 + a_1b_1)x^2 + a_1b_2x^3$. □

Suppose we are given the coefficients $\{a_k\}_{k=0}^{n-1}$ and $\{b_k\}_{k=0}^{n-1}$ of the polynomial $p(x)$ and $q(x)$, respectively, and wish to generate the coefficients $\{c_k\}_{k=0}^{n+m-2}$ of the resulting polynomial product.

The straightforward algorithm for computing each c_k separately from the definition $c_k = \sum_{j=0}^{k} a_j b_{k-j}$ can be done in $O(\log (n+m))$ parallel time, using a total of $\Theta(nm)$ operations. A simple use of the FFT algorithm will reduce the total number of operations to $O((n+m) \log (n+m))$.

We let l be an integer that is a power of 2 such that $n + m - 2 < l \leq 2(n + m - 2)$. We let $\boldsymbol{a} = [a_0, \ldots, a_{l-1}]^T$ and $\boldsymbol{b} = [b_0, \ldots, b_{l-1}]^T$, where $a_j = 0$ for $j > n - 1$ and $b_k = 0$ for $k > m - 1$.

Our algorithm for computing the coefficients $\{c_k\}$ representing the product of the two polynomials $p(x)$ and $q(x)$ consists of the following steps.

- *Step 1:* Compute $\boldsymbol{y} = \boldsymbol{W}_l\boldsymbol{a}$ and $\boldsymbol{z} = \boldsymbol{W}_l\boldsymbol{b}$ using the FFT algorithm. Hence, we now have the values of the polynomials $p(x)$ and $q(x)$ at the points $1, \omega, \omega^2, \ldots, \omega^{l-1}$, where ω is a primitive lth root of unity. More precisely, $y_j = p(\omega^j)$ and $z_j = q(\omega^j)$, where $0 \leq j \leq l - 1$.
- *Step 2:* Compute the values $u_j = y_j z_j$, for all $0 \leq j \leq l - 1$. Clearly, $u_j = p(\omega^j)q(\omega^j) = r(\omega^j)$; hence, we have the values of the polynomial $r(x)$ at the l distinct roots of unity.
- *Step 3:* Compute the inverse DFT of the vector $\boldsymbol{u} = [u_0, u_1, \ldots, u_{l-1}]^T$. The first $n + m - 1$ entries of the resulting vector are equal to the coefficients of the product polynomial $r(x)$.

We therefore have the following theorem.

Theorem 8.8: *The coefficients of the product of two polynomials of degrees $(n - 1)$ and $(m - 1)$, respectively, can be computed in $O(\log (n + m))$ time, using a total of $O((n + m) \log (n + m))$ arithmetic operations.* □

PRAM Model: Since the FFT algorithm can be executed on the arithmetic EREW PRAM, the polynomial multiplication algorithm described here runs on the EREW PRAM within the stated bounds. □

8.4.3 CONVOLUTION USING THE FFT ALGORITHM

A notion closely related to polynomial multiplication is that of the *convolution* of two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$, denoted by $\boldsymbol{a} \otimes \boldsymbol{b}$. If $\boldsymbol{a} = [a_0, \ldots, a_{n-1}]^T$ and $\boldsymbol{b} = [b_0, \ldots, b_{m-1}]^T$, then $\boldsymbol{a} \otimes \boldsymbol{b}$ is defined to be the vector $\boldsymbol{c} = [c_0, \ldots, c_{m+n-1}]^T$, such that $c_k = \sum_{j=0}^{k} a_j b_{k-j}$, where $a_j = 0$ for $j > n - 1$, and $b_j = 0$ for $j > m - 1$.

EXAMPLE 8.11:

If $n = 2$ and $m = 3$, then $\boldsymbol{a} = [a_0, a_1]^T$ and $\mathbf{b} = [b_0, b_1, b_2]^T$. Thus, according to our definition, $\boldsymbol{a} \otimes \boldsymbol{b} = \boldsymbol{c} = [c_0, c_1, c_2, c_3, c_4]^T$, where $c_0 = a_0b_0$, $c_1 = a_0b_1 + a_1b_0$, $c_2 = a_0b_2 + a_1b_1$, $c_3 = a_1b_2$, and $c_4 = 0$. □

Except for the term c_{n+m-1} (which is always equal to 0), the convolution of the two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ is the same as the coefficients of the product of the two polynomials $p(x) = \sum_{j=0}^{n-1} a_j x^j$ and $q(x) = \sum_{j=0}^{m-1} b_j x^j$. This equivalence results in the following corollary to Theorem 8.8.

Corollary 8.2: *The convolution of two vectors of dimensions n and m, respectively, can be computed in* $O(\log (n + m))$ *time, using a total of* $O((n + m) \log(n + m))$ *operations on the EREW PRAM.* □

Closely related to polynomial multiplication and convolution is the class of computations involving Toeplitz matrices. These computations are discussed in the next section.

8.5 Toeplitz Matrices

An $n \times n$ matrix $\boldsymbol{T}$ is called a **Toeplitz matrix** if it satisfies the relation $\boldsymbol{T}(k,l) = \boldsymbol{T}(k-1, l-1)$, for $2 \leq l, k \leq n$. In other words, the entries on each diagonal of $\boldsymbol{T}$ are all equal. Hence, such a matrix is determined by the $2n - 1$ entries appearing in the first row and the first column. We will denote these entries by $t_0, t_1, \ldots, t_{2n-2}$ such that

$$\boldsymbol{T} = \begin{bmatrix} t_{n-1} & t_{n-2} & \cdots & & t_2 & t_1 & t_0 \\ t_n & t_{n-1} & t_{n-2} & \cdots & & t_2 & t_1 \\ t_{n+1} & t_n & t_{n-1} & t_{n-2} & \cdots & & t_2 \\ \vdots & & & \ddots & \ddots & & \vdots \\ \vdots & & & & \ddots & \ddots & \vdots \\ t_{2n-3} & t_{2n-2} & \cdots & & & t_{n-1} & t_{n-2} \\ t_{2n-2} & t_{2n-3} & \cdots & & t_{n+1} & t_n & t_{n-1} \end{bmatrix}. \tag{8.5}$$

We say that the vector $t = [t_0, t_1, \ldots, t_{2n-2}]^T$ *defines* the Toeplitz matrix $\boldsymbol{T}$.

EXAMPLE 8.12:

The 4×4 Toeplitz matrix defined by the vector $[t_0, t_1, t_2, t_3, t_4, t_5, t_6]$ is

$$\boldsymbol{T} = \begin{bmatrix} t_3 & t_2 & t_1 & t_0 \\ t_4 & t_3 & t_2 & t_1 \\ t_5 & t_4 & t_3 & t_2 \\ t_6 & t_5 & t_4 & t_3 \end{bmatrix}.$$

□

Toeplitz matrices occur frequently in numerical computations arising in signal processing. One of our goals in this section is the development of an efficient algorithm for inverting *triangular* Toeplitz matrices, a problem that will be shown to be equivalent to polynomial division.

We start by discussing the problem of multiplying a Toeplitz matrix by a vector, a problem whose solution will be used in our triangular Toeplitz matrix-inversion algorithm.

8.5.1 THE PRODUCT OF A TOEPLITZ MATRIX AND A VECTOR

Let $\boldsymbol{a} = [a_0, a_1, \ldots, a_{n-1}]^T$ be a given vector and let $\boldsymbol{d} = \boldsymbol{Ta}$, where $\boldsymbol{T}$ is an $n \times n$ Toeplitz matrix defined by the vector $\boldsymbol{t} = [t_0, t_1, \ldots, t_{2n-2}]^T$. We are interested in developing an efficient parallel algorithm for computing $\boldsymbol{d}$. The standard algorithm for matrix-vector multiplication computes $\boldsymbol{d}$ in $O(\log n)$ time, using a total of $O(n^2)$ operations. We shall see how to use the convolution algorithm described in Section 8.4 to perform this computation using only $O(n \log n)$ operations.

It is easy to verify that each entry d_l of $\boldsymbol{d}$ is given by $d_l = \sum_{j=0}^{n-1} a_j t_{n+l-j-1}$, for $0 \leq l \leq n-1$. Consider the convolution $\boldsymbol{c}$ of the two vectors $\boldsymbol{a}$ and $\boldsymbol{t}$. Then, according to the definition of the convolution, we have $c_k = \sum_{j=0}^{k} a_j t_{k-j}$. Substituting $k = n + l - 1$, we get $c_{n+l-1} = \sum_{j=0}^{n+l-1} a_j t_{n+l-j-1} = \sum_{j=0}^{n-1} a_j t_{n+l-j-1}$, since $a_j = 0$ for $j > n-1$. Thus, $d_l = c_{n+l-1}$ for $0 \leq l \leq n-1$.

Therefore, we can obtain the product $\boldsymbol{Ta}$ by computing $\boldsymbol{a} \otimes \boldsymbol{t}$, and setting $d_l = c_{n+l-1}$ $(0 \leq l \leq n-1)$.

EXAMPLE 8.13:

For $n = 4$, the components of the vector $\boldsymbol{d} = \boldsymbol{Ta}$ and the entries c_3, c_4, c_5, and c_6 of the convolution $\boldsymbol{c}$ are given by

$$
\begin{aligned}
d_0 = c_3 &= t_3a_0 + t_2a_1 + t_1a_2 + t_0a_3, \\
d_1 = c_4 &= t_4a_0 + t_3a_1 + t_2a_2 + t_1a_3, \\
d_2 = c_5 &= t_5a_0 + t_4a_1 + t_3a_2 + t_2a_3, \\
d_3 = c_6 &= t_6a_0 + t_5a_1 + t_4a_2 + t_3a_3.
\end{aligned}
$$

□

Using Corollary 8.2, we obtain the following lemma.

Lemma 8.1: *We can obtain the product of an $n \times n$ Toeplitz matrix $\boldsymbol{T}$ and a vector $\boldsymbol{a}$ by computing $\boldsymbol{c} = \boldsymbol{a} \otimes \boldsymbol{t}$ and retaining the entries $(c_{n-1}, c_n, \ldots, c_{2n-2})$ of vector $\boldsymbol{c}$, where $\boldsymbol{t}$ is the vector defining $\boldsymbol{T}$. Hence, the matrix product can be obtained in $O(\log n)$ time, using a total of $O(n \log n)$ operations.*

PRAM Model: Since the convolution of two vectors is the only nontrivial routine used, the EREW PRAM model is sufficient to support the algorithm for Toeplitz matrix-vector multiplication within the stated bounds. □

The relationship between Toeplitz matrix-vector product and polynomial multiplication is implicit in the preceding discussion. We can make this relationship clearer as follows.

Let $t(x) = \sum_{i=0}^{r-1} t_i x^i$ and $p(x) = \sum_{i=0}^{m-1} p_i x^i$ be two polynomials. Then, the coefficients of the product polynomial $q(x) = t(x)p(x)$ are given by

$$\begin{bmatrix} q_{r+m-2} \\ q_{r+m-1} \\ \vdots \\ \\ \vdots \\ q_0 \end{bmatrix} = \begin{bmatrix} t_{r-1} & & & & & \\ t_{r-2} & t_{r-1} & & & & \\ \vdots & t_{r-2} & t_{r-1} & \ddots & & \\ t_1 & \vdots & \ddots & \ddots & \ddots & \\ t_0 & t_1 & & \ddots & \ddots & t_{r-1} \\ & t_0 & t_1 & \ddots & \ddots & t_{r-2} \\ & & \ddots & \ddots & \ddots & \vdots \\ & & & \ddots & \ddots & t_1 \\ & & & & & t_0 \end{bmatrix} \begin{bmatrix} p_{m-1} \\ p_{m-2} \\ \vdots \\ \\ \vdots \\ p_0 \end{bmatrix}.$$

The coefficient matrix is an $(r + m - 1) \times m$ Toeplitz matrix. In particular, if we set $r = 2n - 1$ and $m = n$, and remove the top and bottom $(n - 1)$ rows, we get the general Toeplitz matrix introduced in Eq. 8.5. We will later exploit this relationship to perform several important polynomial computations.

EXAMPLE 8.14:

Consider the two polynomials $t(x) = \sum_{i=0}^{4} t_i x^i$ and $p(x) = \sum_{i=0}^{2} p_i x^i$. Then, the coefficients of the product $q(x) = t(x)p(x) = \sum_{i=0}^{6} q_i x^i$ are given by

$$\begin{bmatrix} q_6 \\ q_5 \\ q_4 \\ q_3 \\ q_2 \\ q_1 \\ q_0 \end{bmatrix} = \begin{bmatrix} t_4 & 0 & 0 \\ t_3 & t_4 & 0 \\ t_2 & t_3 & t_4 \\ t_1 & t_2 & t_3 \\ t_0 & t_1 & t_2 \\ 0 & t_0 & t_1 \\ 0 & 0 & t_0 \end{bmatrix} \begin{bmatrix} p_2 \\ p_1 \\ p_0 \end{bmatrix}.$$

Removing the top and the bottom two rows of the Toeplitz matrix, we obtain

$$\begin{bmatrix} q_4 \\ q_3 \\ q_2 \end{bmatrix} = \begin{bmatrix} t_2 & t_3 & t_4 \\ t_1 & t_2 & t_3 \\ t_0 & t_1 & t_2 \end{bmatrix} \begin{bmatrix} p_2 \\ p_1 \\ p_0 \end{bmatrix}.$$

□

8.5.2 INVERSION OF TRIANGULAR TOEPLITZ MATRICES

We now consider a special class of Toeplitz matrices—namely, **triangular Toeplitz** matrices. We develop an efficient parallel scheme for inverting such matrices. Notice that a triangular Toeplitz matrix $\boldsymbol{T}$ is uniquely determined by its first column. We shall exploit this fact to obtain our efficient parallel algorithm.

As we shall see in Section 8.6, *inverting a triangular Toeplitz matrix is equivalent to performing polynomial division*, a fundamental problem in polynomial computations. Again the FFT algorithm will play the crucial role of providing an efficient algorithm.

We have already examined in Section 8.2.1, the problem of computing the inverse of an arbitrary $n \times n$ triangular matrix. The described divide-and-conquer algorithm runs in $O(\log^2 n)$ time, using a total of $O(M(n))$ arithmetic operations. Recall that $M(n)$ is the number of arithmetic operations required for $n \times n$ matrix multiplication. A combination of the divide-and-conquer approach and the FFT algorithm will allow us to invert $n \times n$ triangular Toeplitz matrices in $O(\log^2 n)$ time, using a total of $O(n \log n)$ operations.

Let $\boldsymbol{T}$ be a lower triangular Toeplitz matrix partitioned into $n/2 \times n/2$ blocks,

$$\boldsymbol{T} = \begin{bmatrix} \boldsymbol{T}_1 & 0 \\ \boldsymbol{T}_2 & \boldsymbol{T}_1 \end{bmatrix},$$

where n is assumed to be a power of 2. The two blocks on the diagonal are identical, and the blocks $\boldsymbol{T}_1$ and $\boldsymbol{T}_2$ are Toeplitz matrices. As we saw in Section 8.2.1, $\boldsymbol{T}^{-1}$ is given by

$$\boldsymbol{T}^{-1} = \begin{bmatrix} \boldsymbol{T}_1^{-1} & 0 \\ -\boldsymbol{T}_1^{-1}\boldsymbol{T}_2\boldsymbol{T}_1^{-1} & \boldsymbol{T}_1^{-1} \end{bmatrix}.$$

Since $\boldsymbol{T}^{-1}$ is also a lower triangular Toeplitz matrix, $\boldsymbol{T}^{-1}$ is uniquely defined by its first column. Based on the preceding identity, we deduce the following algorithm for computing the first column of $\boldsymbol{T}^{-1}$:

- *Step 1:* Recursively compute the first column of T_1^{-1}.
- *Step 2:* Compute the first column of $-T_1^{-1}T_2T_1^{-1}$.

The computation of the first column of $-T_1^{-1}T_2T_1^{-1}$ is equivalent to computing $-T_1^{-1}T_2T_1^{-1}e$, where $e = [1, 0, \ldots, 0]^T$. We can achieve such a computation successively by computing $T_1^{-1}e$, $T_2(T_1^{-1}e)$, and $-T_1^{-1}(T_2(T_1^{-1}e))$. Once the first column of T_1^{-1} has been computed, we can use the algorithm presented in Section 8.5.1 for multiplying a Toeplitz matrix by a vector to compute the first column of $-T_1^{-1}T_2T_1^{-1}$ in $O(\log n)$ time, using a total of $O(n \log n)$ arithmetic operations.

Therefore, the running time $T(n)$ and the number of arithmetic operations $W(n)$ used by our algorithm for inverting an $n \times n$ lower triangular Toeplitz matrix satisfy the following recurrences:

$$T(n) = T(n/2) + O(\log n)$$
$$W(n) = W(n/2) + O(n \log n).$$

It is simple to verify that the solutions to these recurrences are $T(n) = O(\log^2 n)$ and $W(n) = O(n \log n)$. Hence, we have shown the following theorem.

Theorem 8.9: *We can invert an $n \times n$ triangular Toeplitz matrix in $O(\log^2 n)$ time, using a total of $O(n \log n)$ arithmetic operations.* □

PRAM Model: The algorithm for inverting a triangular Toeplitz matrix described in this section can be viewed as a series of convolution computations. Therefore, the algorithm runs on the arithmetic EREW PRAM model. □

8.5.3 *CIRCULANT MATRICES

Another important special class of Toeplitz matrices consists of the matrices C defined by a vector $a = [a_0, a_1, \ldots, a_{n-1}]^T$ such that

$$\mathbf{C} = \begin{bmatrix} a_0 & a_1 & & \cdots & a_{n-1} \\ a_{n-1} & a_0 & a_1 & \cdots & a_{n-2} \\ a_{n-2} & a_{n-1} & a_0 & \cdots & a_{n-3} \\ \vdots & & \ddots & \ddots & \vdots \\ a_2 & a_3 & \cdots & a_{n-1} & a_0 \end{bmatrix}.$$

Such a matrix C is called a **circulant matrix.** We shall show that we can compute the inverse of a circulant matrix very quickly by using the FFT algorithm. The following fact is the basis of our algorithm; its proof is left to Exercise 8.16.

Theorem 8.10: *Let C be an $n \times n$ circulant matrix defined by the vector a. Then, $W_n^{-1} C W_n = D$, where D is a diagonal matrix such that $d = [d_{11}, d_{22}, \ldots, d_{nn}]^T$ is the DFT of the vector a, and W_n is the coefficient matrix for the DFT of dimension n. In particular, the components of the DFT of a and the columns of W_n are, respectively, the eigenvalues and the eigenvectors of C.* □

Our circulant matrix-inversion algorithm will make use of an algorithm for solving linear systems whose coefficient matrices are circulant. We start by discussing the latter problem.

Suppose we wish to solve the linear system $Cx = b$. Using Theorem 8.10, we obtain that $x = W_n D^{-1} W_n^{-1} b$. Therefore, we can determine x by the following algorithm:

- *Step 1:* Compute $b' = W_n^{-1} b$, the inverse DFT of the vector b that defines the right-hand side of the linear system.
- *Step 2:* Compute $d = W_n a$, the DFT of the vector a that defines the circulant matrix C.
- *Step 3:* Compute $b'' = D^{-1} b'$, where D is a diagonal matrix defined by the vector d.
- *Step 4:* Compute $W_n b''$, the DFT of the vector generated at step 3.

It is clear that steps 1, 2, and 4 can be done using the FFT algorithm; the third step consists of a set of n independent multiplications (actually, divisions). Therefore, the linear system $Cx = b$ can be solved in $O(\log n)$ time, using a total of $O(n \log n)$ operations.

To compute the inverse of a circulant matrix C, we observe that C^{-1} is also a circulant matrix and hence is uniquely determined by its first column. Therefore, solving the linear system $Cx = e$, where $e = [1, 0, \ldots, 0]^T$ is sufficient to determine C^{-1}.

We therefore have shown the following theorem.

Theorem 8.11: *Inversion of an $n \times n$ circulant matrix can be performed in $O(\log n)$ time, using a total of $O(n \log n)$ operations.* □

PRAM Model: As before, the arithmetic EREW PRAM model is sufficient, since the only nontrivial routine used is the FFT algorithm. □

8.6 Polynomial Division

Consider the two polynomials $s(x) = \sum_{j=0}^{n-1} s_j x^j$ and $t(x) = \sum_{j=0}^{m-1} t_j x^j$. Then there exist unique polynomials, the **quotient** $q(x)$ and the **remainder** $r(x)$, such

that $s(x) = t(x)q(x) + r(x)$, and $\deg(r(x)) < \deg(t(x))$, where $\deg(u(x))$ denotes the degree of the polynomial $u(x)$. Here, we are assuming that the coefficients come from a commutative ring that can support the FFT algorithm, and that the coefficient t_{m-1} has a multiplicative inverse in the ring. The reader may wish to assume that all the coefficients are complex numbers.

EXAMPLE 8.15:

Let $s(x) = x^7 - 5x^3 + x^2 - 1$ and $t(x) = x^4 + 2x^3 - x^2 - 2x$. Then, the quotient $q(x)$ and the remainder $r(x)$ are given by $q(x) = x^3 - 2x^2 + 5x - 10$ and $r(x) = 16x^3 + x^2 - 20x - 1$. □

In this section, we shall examine an efficient algorithm for computing the coefficients of the polynomials $q(x)$ and $r(x)$ given the coefficients of $s(x)$ and $t(x)$. This problem is clearly a fundamental one that has been examined by many researchers through the years. We shall provide a solution based on the algorithm for inverting triangular Toeplitz matrices given in Section 8.5.2.

We start our description of the algorithm to compute $q(x)$ and $r(x)$ by showing that we need to compute only one of the two polynomials $q(x)$ and $r(x)$.

Lemma 8.2: *Let $t(x)$ and $s(x)$ be two polynomials of degrees $n - 1$ and $m - 1$, respectively, where $n > m$. If the quotient $q(x)$ is known, $r(x)$ can be computed in $O(\log n)$ time, using a total of $O(n \log n)$ operations. Conversely, if $r(x)$ is known, $q(x)$ can be computed in $O(\log (n - m))$ time, using a total of $O(n + (n - m) \log (n - m))$ operations.*

Proof: Let $q(x)$ be known. Clearly, $\deg q(x) = n - m$. We can multiply the two polynomials $t(x)q(x)$ in $O(\log n)$ time, using $O(n \log n)$ operations, using the FFT-based algorithm. Computing the remainder $r(x) = s(x) - t(x)q(x)$ can then be done in $O(1)$ steps, using $O(n)$ operations. It follows that, once we know the quotient $q(x)$, the remainder $r(x)$ can be computed in $O(\log n)$ time, using a total of $O(n \log n)$ operations.

Suppose now that $r(x)$ is known. Then, $s(x) - r(x)$ can be computed in $O(1)$ steps, using a total of $O(n)$ operations. We can determine the polynomial $q(x) = \frac{s(x) - r(x)}{t(x)}$ by exploiting the connection between polynomial evaluation/interpolation and DFT. More precisely, we compute $s(\omega^l) - r(\omega^l)$ and $t(\omega^l)$ for $0 \leq l < n - m + 1$, where ω is a primitive $(n - m + 1)$st root of unity. The FFT algorithm can be used to achieve this computation in $O(\log (n - m))$ time, using a total of $O((n - m) \log (n - m))$ operations. Hence, $q(\omega^l)$ can be computed within these bounds. The coefficients of $q(x)$ can then be determined by the inverse FFT algorithm. Hence, once we know the remainder $r(x)$, the quotient $q(x)$ can be computed in $O(\log (n - m))$ time, using $O(n + (n - m) \log (n - m))$ operations. □

We conclude from Lemma 8.2 that the division of two polynomials can be reduced to computing either the quotient $q(x)$ or the remainder $r(x)$. We concentrate on the computation of the quotient $q(x)$.

Let $q(x) = \sum_{j=0}^{n-m} q_j x^j$ be the quotient of the two polynomials $s(x) = \sum_{j=0}^{n-1} s_j x^j$ and $t(x) = \sum_{j=0}^{m-1} t_j x^j$. Without loss of generality, we assume that $t_{m-1} s_{n-1} \neq 0$. The product $t(x)q(x)$ can be expressed as $\sum_{k=0}^{n-1} a_k x^k$, where $a_k = \sum_{j=0}^{k} t_j q_{k-j}$.

We let $r(x)$ be the remainder of the division of $s(x)$ by $t(x)$. Given that deg $r(x) < m - 1$, we conclude that $\sum_{k=m-1}^{n-1} s_k x^k = \sum_{k=m-1}^{n-1} a_k x^k$, and thus $s_k = a_k$, for $m - 1 \leq k \leq n - 1$. It follows that $s_k = \sum_{j=0}^{k} t_j q_{k-j}$, when $m - 1 \leq k \leq n - 1$. Using that $t_j = 0$ for $j \geq m$ and $q_j = 0$ for $j \geq n - m + 1$, we obtain the following identities:

$$
\begin{aligned}
s_{n-1} &= t_{m-1} q_{n-m} \\
s_{n-2} &= t_{m-2} q_{n-m} + t_{m-1} q_{n-m-1} \\
s_{n-3} &= t_{m-3} q_{n-m} + t_{m-2} q_{n-m-1} + t_{m-1} q_{n-m-2} \\
s_{n-4} &= t_{m-4} q_{n-m} + t_{m-3} q_{n-m-1} + t_{m-2} q_{n-m-2} + t_{m-1} q_{n-m-3} \\
&\vdots \\
&.
\end{aligned}
$$

In matrix form, we have

$$
\begin{bmatrix} s_{n-1} \\ s_{n-2} \\ s_{n-3} \\ \vdots \\ s_{m-1} \end{bmatrix} = \begin{bmatrix} t_{m-1} & 0 & 0 & 0 & \cdots & 0 \\ t_{m-2} & t_{m-1} & 0 & 0 & \cdots & 0 \\ t_{m-3} & t_{m-2} & t_{m-1} & 0 & \cdots & 0 \\ t_{m-4} & t_{m-3} & t_{m-2} & t_{m-1} & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots \end{bmatrix} \begin{bmatrix} q_{n-m} \\ q_{n-m-1} \\ q_{n-m-2} \\ \vdots \\ q_0 \end{bmatrix}.
$$

The matrix $\boldsymbol{T}$ relating the coefficients of $s(x)$ and $q(x)$ is an $(n - m + 1) \times (n - m + 1)$ lower triangular Toeplitz matrix. This relationship should come as no surprise in view of the close connection between Toeplitz matrices and polynomial multiplication.

EXAMPLE 8.16:

Let $s(x)$ be of degree 8, and let $t(x)$ be of degree 4. Then the coefficients of $q(x)$ are defined by the following linear system of equations:

$$
\begin{bmatrix} s_7 \\ s_6 \\ s_5 \\ s_4 \\ s_3 \end{bmatrix} = \begin{bmatrix} t_3 & 0 & 0 & 0 & 0 \\ t_2 & t_3 & 0 & 0 & 0 \\ t_1 & t_2 & t_3 & 0 & 0 \\ t_0 & t_1 & t_2 & t_3 & 0 \\ 0 & t_0 & t_1 & t_2 & t_3 \end{bmatrix} \begin{bmatrix} q_4 \\ q_3 \\ q_2 \\ q_1 \\ q_0 \end{bmatrix}.
$$

□

We see that the coefficients of $q(x)$ are given by the solution to a linear system defined by an $(n - m + 1) \times (n - m + 1)$ lower triangular Toeplitz matrix. Using Theorem 8.9, we immediately conclude the following theorem.

Theorem 8.12: *The division of two polynomials of degrees n and m, respectively, where $n > m$, can be done in $O(\log^2 (n - m) + \log n)$ time, using a total of $O(n \log n)$ operations.* □

PRAM Model: The polynomial-division algorithm is deduced from the algorithm for solving a triangular Toeplitz linear system of equations. Therefore, we need only the arithmetic EREW PRAM model. □

8.7 Polynomial Evaluation and Interpolation

In this section, we consider the problem of evaluating a polynomial $p(x)$ of degree $n - 1$ at n distinct points, and the dual problem of interpolating a polynomial at n distinct points. We have already come across these two problems in the case where the n distinct points are roots of unity (Section 8.4). This section deals with the general case of an arbitrary set of n distinct points.

8.7.1 POLYNOMIAL EVALUATION

Let $p(x) = \sum_{j=0}^{n-1} a_j x^j$ be an $(n - 1)$st-degree polynomial given by its coefficients, and let $\{\alpha_l \mid 0 \leq l \leq n - 1\}$ be a set of n *distinct* points. We wish to compute $p(\alpha_l)$ for $0 \leq l \leq n - 1$.

One way of evaluating the polynomial $p(x)$ at the given n distinct points would be to compute all the $p(\alpha_l)$'s concurrently, using Horner's algorithm for each such evaluation. The total running time would be $O(\log n)$, and the total number of arithmetic operations would be $O(n^2)$. However, the best known sequential algorithm uses $O(n \log^2 n)$ arithmetic operations. Using the FFT algorithm (Algorithm 8.2) and the polynomial-division algorithm (Section 8.6), we will be able to provide a fast parallel implementation of the sequential algorithm without increasing the total number of operations.

Let $q_l = x - \alpha_l$, where $0 \leq l \leq n - 1$. We shall view polynomial evaluation at α_l as the computation of the remainder of the division of $p(x)$ by $q_l(x)$; that is, $p(\alpha_l) = p(x) \bmod q_l(x)$. It is not clear at first sight what this alternate view has to offer. We now proceed to illustrate its usefulness.

Let $Q_1(x) = \prod_{l=0}^{\frac{n}{2}-1} q_l(x)$ and $Q_2(x) = \prod_{l=\frac{n}{2}}^{n-1} q_l(x)$, where n is assumed to be a power of 2. $Q_1(x)$ and $Q_2(x)$ are polynomials, each of degree $\frac{n}{2}$. Let $p_1(x) = p(x) \bmod Q_1(x)$ and $p_2(x) = p(x) \bmod Q_2(x)$. Hence, $p_1(x)$ and $p_2(x)$ are the remainders of the division of $p(x)$ by $Q_1(x)$ and $Q_2(x)$, respectively. Thus, $p(x) = t_1(x)Q_1(x) + p_1(x)$ and $p(x) = t_2(x)Q_2(x) + p_2(x)$, for some polynomials $t_1(x)$ and $t_2(x)$, and $\deg p_1(x)$, $\deg p_2(x) < \frac{n}{2}$. The crucial observation that allows the development of an efficient algorithm is that $p(\alpha_l) = p_1(\alpha_l)$, for $0 \le l \le \frac{n}{2} - 1$, and that $p(\alpha_l) = p_2(\alpha_l)$, for $\frac{n}{2} \le l \le n - 1$. That is, the first set of the desired $\frac{n}{2}$ values of $p(x)$ consists of the values of the polynomial $p_1(x)$ of degree $\frac{n}{2}$ at the points $\alpha_0, \alpha_1, \ldots, \alpha_{\frac{n}{2}-1}$, and the second set of the desired $\frac{n}{2}$ values of $p(x)$ consists of the values of $p_2(x)$ of degree $\frac{n}{2}$ at the points $\alpha_{\frac{n}{2}}$, $\alpha_{\frac{n}{2}+1}, \ldots, \alpha_{n-1}$.

We can now apply the same process to each of $p_1(x)$ and $p_2(x)$. The overall procedure will be given next. The algorithm is described in terms of a forward traversal of a balanced binary tree to compute all the necessary q_l products. These products are denoted by $Q_{h,j}(x)$, where the polynomial $Q_{h,j}(x)$ is generated at the jth node of height h and the nodes are numbered left to right. Once $\prod_{l=0}^{\frac{n}{2}-1} q_l(x)$ and $\prod_{l=\frac{n}{2}}^{n-1} q_l(x)$ are computed, we traverse the binary tree backward, reducing each polynomial at a given node modulo the products stored at the children. The nodes are numbered by pairs of integers (h,j), where $0 \le h \le \log n$, and $0 \le j \le (n/2^h) - 1$ (see Fig. 8.3 for the case when $n = 8$).

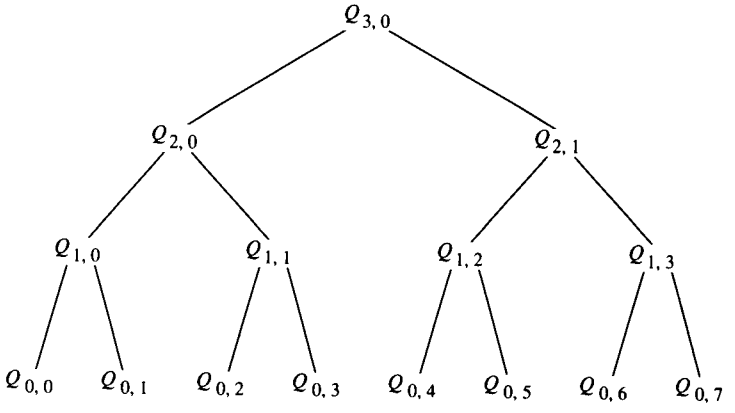


FIGURE 8.3
The binary tree used for polynomial evaluation.

ALGORITHM 8.3
(Polynomial Evaluation)

Input: *(1) A polynomial $p(x)$ of degree $n - 1$ specified by its coefficients, and (2) a set of n distinct points α_j, where $0 \le j \le n - 1$ and $n = 2^k$ for some integer k.*

Output: *The values $p(\alpha_j)$, where $0 \le j \le n - 1$.*

begin

1. **for** $0 \le j \le n - 1$ **pardo**
   Set $Q_{o,j}(x) := x - \alpha_j$
2. **for** $h = 1$ **to** $\log n - 1$ **do**
   for $0 \le j \le (n/2^h) - 1$ **pardo**
   Set $Q_{h,j}(x) := Q_{h-1,2j}(x) \times Q_{h-1,2j+1}(x)$
3. *Set* $p_{k,0}(x) := p(x)$
4. **for** $h = \log n - 1$ **to** 0 **do**
   for $0 \le j \le (n/2^h) - 1$ **pardo**
   j even : Set $p_{h,j}(x) := p_{h+1,\frac{j}{2}}(x) \bmod Q_{h,j}(x)$
   j odd : Set $p_{h,j}(x) := p_{h+1,\frac{j-1}{2}}(x) \bmod Q_{h,j}(x)$
5. **for** $0 \le j \le n - 1$ **pardo**
   Set $p(\alpha_j) := p_{0,j}$

end

EXAMPLE 8.17:

Let $p(x) = x^7 - 5x^3 + x^2 - 1$, and let $\alpha_0 = 0, \alpha_1 = 1, \alpha_2 = -1, \alpha_3 = -2, \alpha_4 = 2, \alpha_5 = 3, \alpha_6 = -3$, and $\alpha_7 = 5$. At step 1 of Algorithm 8.3, we initialize the following quantities: $Q_{0,0} = x, Q_{0,1} = x - 1, Q_{0,2} = x + 1, Q_{0,3} = x + 2, Q_{0,4} = x - 2, Q_{0,5} = x - 3, Q_{0,6} = x + 3$, and $Q_{0,7} = x - 5$. During the first iteration of the loop at step 2, we compute $Q_{1,0} = x(x - 1) = x^2 - x$, $Q_{1,1} = (x + 1)(x + 2) = x^2 + 3x + 2, Q_{1,2} = (x - 2)(x - 3) = x^2 - 5x + 6$, and $Q_{1,3} = (x + 3)(x - 5) = x^2 - 2x - 15$. The second iteration generates $Q_{2,0} = x^4 + 2x^3 - x^2 - 2x$, and $Q_{2,1} = x^4 - 7x^3 + x^2 + 63x - 90$. We then set $p_{3,0} = p(x)$ at step 3. At step 4, we start by computing $p_{2,0} = p(x) \bmod Q_{2,0}(x) = 16x^3 + x^2 - 20x - 1$, and computing $p_{2,1} = p(x) \bmod Q_{2,1}(x)$. We now concentrate on the subtree rooted at $(2, 0)$. For $h = 1$, we get $p_{1,0} = p_{2,0}(x) \bmod Q_{2,0}(x) = -3x - 1$, and $p_{1,1} = p_{2,0} \bmod Q_{1,1}(x) = 89x + 93$. Finally, we generate the polynomials at the leaves, $p_{0,0} = p_{1,0}(x) \bmod Q_{0,0}(x) = -1, p_{0,1} = p_{1,0}(x) \bmod Q_{0,1}(x) = -4, p_{0,2} = p_{1,1}(x) \bmod Q_{0,2}(x) = 4$, and $p_{0,3} = p_{1,1}(x) \bmod Q_{0,3}(x) = -85$. We perform a similar process for the children of $Q_{2,1}(x)$. □

Theorem 8.13: *Algorithm 8.3 correctly computes the values of the given polynomial at the n distinct points. It can be implemented to run in $O(\log^3 n)$ parallel time, using a total of $O(n \log^2 n)$ operations.*

Proof: The correctness proof follows from the discussion preceding the statement of the theorem. The complexity bounds can be estimated as follows.

Step 1 takes $O(1)$ time and uses $O(n)$ operations. The loop at step 2 consists of $O(\log n)$ iterations such that the hth iteration involves $n/2^h$ polynomial multiplications, where each polynomial is of degree 2^{h-1}. The multiplication of two polynomials, each of degree 2^{h-1}, can be performed in $O(h)$ time, using $O(h2^h)$ operations. Therefore, each iteration of the loop at step 2 can be completed in $O(\log n)$ time, using $O((n/2^h)h2^h) = O(n \log n)$ operations. It follows that step 2 can be executed in $O(\log^2 n)$ time, using a total of $O(n \log^2 n)$ operations.

Step 3 takes $O(1)$ time, using $O(n)$ operations. Step 4 consists of $\log n$ iterations such that each iteration requires $n/2^h$ polynomial divisions, each division being between a polynomial of degree at most 2^{h+1} and a polynomial of degree 2^h, where h is the index appearing in the loop. Hence, each iteration can be performed in $O(\log^2 h) = O(\log^2 n)$ time, using a total of $O((n/2^h)2^h h) = O(n \log n)$ operations. Therefore, step 4 takes $O(\log^3 n)$ time and uses a total of $O(n \log^2 n)$ operations.

Finally, step 5 is straightforward and can be done in $O(1)$ time, using $O(n)$ operations. Hence, the theorem follows. □

PRAM Model: The scheme of Algorithm 8.3 is based on a forward and a backward traversal of a balanced binary tree. The main routines used at each node are those for polynomial multiplication (Section 8.4) and polynomial division (Section 8.6). Therefore, Algorithm 8.3 runs on the EREW PRAM model. □

Remark 8.1: We mentioned, in Section 8.4, that the computation of the DFT of a vector $\boldsymbol{a}$ is equivalent to polynomial evaluation at the n roots of unity, where the list of coefficients of the polynomial is defined by $\boldsymbol{a}$. We can use Algorithm 8.3 as the basis for deriving a fast algorithm for computing the DFT by exploiting the fact that the n distinct points α_j are the n distinct roots of unity. The details are left to Exercise 8.20. □

8.7.2 POLYNOMIAL INTERPOLATION

We now discuss the dual problem of determining the coefficients of the polynomial $p(x)$ of degree $n-1$ given by the set of the latter's values $\{\beta_j = p(\alpha_j)\}_{j=0}^{n-1}$, where the α_j's are distinct. The classical **Lagrange interpolation formula** specifies $p(x)$ as follows:

$$p(x) = \sum_{j=0}^{n-1} \beta_j \frac{\prod_{l=0, l\neq j}^{n-1}(x-\alpha_j)}{\prod_{l=0, l\neq j}^{n-1}(\alpha_j-\alpha_l)} \tag{8.6}$$

It is easy to verify that $p(x)$ takes on the appropriate values at the n distinct points $\alpha_0, \alpha_1, \ldots, \alpha_{n-1}$. This formula looks computationally unattractive. We will rearrange the terms so as to make this formula much more useful.

As in the case of polynomial evaluation, we define the linear polynomials $q_l(x) = x - \alpha_l$, and we let $Q(x) = \prod_{l=0}^{n-1} q_l(x)$. The derivative $Q'(x)$ is a polynomial of degree $n - 1$ that can be obtained easily from the polynomial $Q(x)$ when given by the list of its coefficients. In addition, $Q'(\alpha_j) = \prod_{l=0, l\neq j}^{n-1}(\alpha_j - \alpha_l)$.

Therefore, we can obtain the denominators appearing in Eq. 8.6 by evaluating the polynomial $Q'(x)$ at the points α_j, where $0 \leq k \leq n - 1$. We can use Algorithm 8.3 to obtain these quantities. Hence, assume we have all the values $\gamma_j = Q'(\alpha_j)$, for $0 \leq j \leq n - 1$.

We let $c_j = \beta_j/\gamma_j$, where $0 \leq j \leq n - 1$. Then, Eq. 8.6 can be rewritten as follows:

$$p(x) = Q(x)\sum_{j=0}^{n-1}\frac{c_j}{x-\alpha_j}. \tag{8.7}$$

Therefore, the desired polynomial $p(x)$ is the numerator of the sum $\sum_{j=0}^{n-1}\frac{c_j}{x-\alpha_j}$. It turns out that we can compute this sum easily by using a balanced binary tree method, as in Algorithm 8.3. Using the same notation for the nodes of the binary tree, let $Q_{k-1,0}(x) = \prod_{l=0}^{\frac{n}{2}-1} q_l(x)$, and $Q_{k-1,1}(x) = \prod_{l=\frac{n}{2}}^{n-1} q_l(x)$, where $n = 2^k$.

We denote the numerators of the sums $\sum_{j=0}^{\frac{n}{2}-1}\frac{c_j}{x-\alpha_j}$ and $\sum_{j=\frac{n}{2}}^{n-1}\frac{c_j}{x-\alpha_j}$ by $p_{k-1,0}(x)$ and $p_{k-1,1}(x)$, respectively. Clearly, this definition gives us

$$\begin{aligned}\frac{p(x)}{Q(x)} &= \frac{p_{k-1,0}(x)}{Q_{k-1,0}(x)} + \frac{p_{k-1,1}(x)}{Q_{k-1,1}(x)} \\ &= \frac{p_{k-1,0}(x)Q_{k-1,1}(x) + p_{k-1,1}(x)Q_{k-1,0}(x)}{Q(x)}.\end{aligned}$$

It follows that $p(x) = p_{k-1,0}(x)Q_{k-1,1}(x) + p_{k-1,1}(x)Q_{k-1,0}(x)$. Precisely the same procedure can be used to compute the polynomials $p_{k-1,0}(x)$ and $p_{k-1,1}(x)$. The detailed algorithm follows.

ALGORITHM 8.4

(Polynomial Interpolation)

Input: *A set of pairs of numbers* (α_j, β_j), *where* $0 \leq j \leq n$, *the* α_j*'s are distinct, and* $n = 2^k$ *for some integer* k.

Output: *The coefficients of the* $(n - 1)$*st-degree polynomial* $p(x)$ *such that* $p(\alpha_j) = \beta_j$, *for all* $0 \leq j \leq n - 1$.

begin

1. **for** $0 \leq j \leq n - 1$ **pardo**
   Set $Q_{0,j}(x) := x - \alpha_j$

2. **for** $h = 1$ **to** $\log n$ **do**
   for $0 \le j \le (n/2^h) - 1$ **pardo**
   Set $Q_{h,j}(x)$: $= Q_{h-1,2j}(x) \times Q_{h-1,2j+1}(x)$
3. *Compute* $Q'_{k,0}(x)$, *and use Algorithm 8.3 to compute* $\gamma_j = Q'_{k,0}(\alpha_j)$, *for all* $0 \le j \le n - 1$.
4. **for** $0 \le j \le n - 1$ **pardo**
   Set $p_{0,j}(x)$: $= \beta j/\gamma_j$
5. **for** $h = 1$ **to** $\log n$ **do**
   for $0 \le j \le (n/2^h) - 1$ **pardo**
   Set $p_{h,j}(x)$: $= p_{h-1,2j}(x)Q_{h-1,2j+1}(x) + p_{h-1,2j+1}(x)Q_{h-1,2j}(x)$
6. *Set* $p(x)$: $= p_{k,0}(x)$.

end

EXAMPLE 8.18:

Suppose that we want to determine the coefficients of the polynomial defined at the points $(0, -1)$, $(1, -4)$, $(-1, 4)$, and $(-2, -85)$. After step 1 is completed, we obtain $Q_{0,0} = x$, $Q_{0,1} = x - 1$, $Q_{0,2} = x + 1$, and $Q_{0,3} = x + 2$. The loop defined at step 2 runs twice. During the first iteration, the two polynomials $Q_{1,0} = x(x - 1) = x^2 - x$ and $Q_{1,1} = (x + 1)(x + 2) = x^2 + 3x + 2$ are generated. The polynomial $Q_{2,0} = x^4 + 2x^3 - x^2 - 2x$ is generated at the second iteration. The derivative of $Q_{2,0}$ is given by $Q'_{2,0} = 4x^3 + 6x^2 - 2x - 2$. The values of $Q'_{2,0}$ at the points 0, 1, -1, and -2 yield $\gamma_0 = -2$, $\gamma_1 = 6$, $\gamma_2 = 2$, and $\gamma_3 = -6$. In step 4, we compute the numbers $p_{0,0} = \frac{1}{2}$, $p_{0,1} = -\frac{2}{3}$, $p_{0,2} = 2$, and $p_{0,3} = \frac{85}{6}$. The loop at step 5 runs for two iterations, the first of which produces $p_{1,0} = p_{0,0}Q_{0,1} + p_{0,1}Q_{0,0} = \frac{-x-3}{6}$, and $p_{1,1} = p_{0,2}Q_{0,3} + p_{0,3}Q_{0,2} = \frac{97x+109}{6}$. At the end of the second iteration, we compute $p(x) = p_{2,0} = p_{1,0}Q_{1,1} + p_{1,1}Q_{1,0} = 16x^3 + x^2 - 20x - 1$. □

Theorem 8.14: *Determining the coefficients of an* $(n - 1)$*st-degree polynomial specified by its values at n distinct points can be achieved in* $O(\log^3 n)$ *time, using a total of* $O(n \log^2 n)$ *operations.*

Proof: The proof of the following theorem is similar to that of Theorem 8.13. The details are left to the reader. □

PRAM Model: Like the algorithm for polynomial evaluation (Algorithm 8.3), Algorithm 8.4 runs on the EREW PRAM model within the stated bounds. □

8.8 General Dense Matrices

So far in this chapter, we have encountered matrix problems involving either triangular or other special matrices, such as Toeplitz and circulant matrices. The algorithms described for inverting an $n \times n$ circulant matrix and an $n \times n$ triangular Toeplitz matrix are efficient, since the total number of operations used in each case is $O(n \log n)$. Our algorithm for inverting an $n \times n$ *arbitrary* triangular matrix is disappointing in this regard, since it uses $O(M(n))$ operations, compared to $O(n^2)$ operations used by the standard forward-substitution sequential algorithm. The algorithms to be described in this section for handling general dense matrices achieve polylogarithmic running time, but use considerably more operations than those required by the best known sequential algorithms.

The methods to which we just referred, called **direct methods,** should be contrasted with the class of **iterative methods** for which efficient parallel implementations *do exist*. Iterative methods consist of improving successive approximations until a close enough solution is found. On our arithmetic PRAM model, iterative methods can generate only approximate solutions, but we will insist that these solutions be very close to the exact solutions in a certain sense.

We shall start by describing fast direct methods for handling basic dense-matrix computations, followed by a description of an efficient parallel iterative algorithm for computing the inverse of a nonsingular matrix.

8.8.1 DIRECT METHODS

Let A be an $n \times n$ matrix whose entries come from an arbitrary field $\mathcal{F}$, and let λ be an indeterminate. The **characteristic polynomial** of A is defined to be $\phi(\lambda) = \det(\lambda I - A) = \sum_{i=0}^{n} c_i \lambda^i$, which is an nth-degree polynomial whose roots are called the **eigenvalues** of A. In particular, this definition implies that $\phi(0) = c_0 = \det(-A)$, and hence $\det(A) = (-1)^n c_0$.

The characteristic polynomial plays an important role in determining canonical forms of A under the *similarity transformation*. For our purposes, the importance of the characteristic polynomial is due partially to the following well-known theorem from linear algebra.

Theorem 8.15 (Cayley–Hamilton): *Let A be a matrix whose entries come from an arbitrary field, and let $\phi(\lambda) = \sum_{i=0}^{n} c_i \lambda^i$ be its characteristic polynomial. Then, $\phi(A) = \sum_{i=0}^{n} c_i A^i = 0$.* □

EXAMPLE 8.19:

Let A be the 3×3 matrix

$$\begin{bmatrix} -1 & 2 & 0 \\ 3 & 2 & 1 \\ 0 & 0 & -1 \end{bmatrix}.$$

The matrix $\lambda I - A$ is given by

$$\begin{bmatrix} \lambda + 1 & -2 & 0 \\ -3 & \lambda - 2 & -1 \\ 0 & 0 & \lambda + 1 \end{bmatrix},$$

whose determinant is $\phi(\lambda) = \lambda^3 - 9\lambda - 8$. The Cayley-Hamilton theorem (Theorem 8.13) states that $A^3 - 9A - 8I = 0$, as can be easily verified. In addition, $\phi(0) = -8$, and thus $\det(A) = 8$. □

For the remainder of this section, we will assume that our field $\mathcal{F}$ has *characteristic* 0; that is, given any element $a \in \mathcal{F}$, the relation $a + a + \cdots + a = 0$ implies $a = 0$.

The following lemma indicates the relevance of the Cayley-Hamilton theorem (Theorem 8.13) to the problem of solving a linear system of equations.

Lemma 8.3: *Given the characteristic polynomial $\phi(\lambda)$ of an $n \times n$ matrix A, the linear system of equations defined by $Ax = b$ can be solved in $O(\log^2 n)$ time, using a total of $O(M(n) \log n)$ operations.*

Proof: Let the characteristic function be given by $\phi(\lambda) = \sum_{i=0}^{n} c_i \lambda^i$. By the Cayley-Hamilton theorem, we have $\sum_{i=1}^{n} c_i A^i = -c_0 I$. This equation implies that, for an arbitrary vector x, we have $\sum_{i=1}^{n} c_i A^i x = -c_0 x$. In particular, if x is such that $Ax = b$, we get $A^i x = A^{i-1}(Ax) = A^{i-1} b$, and thus $\sum_{i=1}^{n} c_i A^{i-1} b = c_0 x$.

Recall from Example 8.3 that the Krylov matrix associated with an $n \times n$ matrix A and an n-dimensional vector b is the matrix $[b, Ab, A^2 b, \ldots, A^{n-1} b]$. We have seen that such a matrix can be computed in $O(\log^2 n)$ time, using $O(M(n) \log n)$ operations (Example 8.3). Since the characteristic polynomial is given, the coefficients c_i are known, and hence $\sum_{i=1}^{n} c_i A^{i-1} b$ is just a linear combination of the columns of the Krylov matrix. If $c_0 = 0$, we declare the system to be singular; otherwise, we set $x = -\frac{1}{c_0} \sum_{i=1}^{n} c_i A^{i-1} b$. □

A partial converse of Lemma 8.3 exists (see Exercise 8.21).

Implicit in the proof of Lemma 8.3 is a method for computing A^{-1}, whenever it exists, from the polynomial $\phi(\lambda)$. In fact, the Cayley-Hamilton theorem implies $(\sum_{i=1}^{n} c_i A^{i-1}) A = -c_0 I$. The matrix A is invertible if and

only if $c_0 \neq 0$ (recall also that $\det(A) = (-1)^n c_0$). And, in this case, A^{-1} is given by $A^{-1} = -\frac{1}{c_0}\sum_{i=1}^{n} c_i A^{i-1}$.

We have thus far shown that the coefficients of the characteristic polynomial of a matrix A can be used to solve a linear system of equations whose coefficient matrix is defined by A, and can be used to invert A whenever A is nonsingular. In addition, $\det(A) = (-1)^n c_0$, where c_0 is the constant coefficient of the characteristic polynomial. We now consider the problem of developing a fast parallel algorithm for computing the characteristic polynomial.

Computation of the Characteristic Polynomial. We define the **trace** of a matrix A to be $tr(A) = \sum_{i=1}^{n} a_{ii}$. The trace is simply the sum of the entries on the diagonal of A. Let $s_k = tr(A^k)$, for $1 \leq k \leq n-1$. The coefficients c_i of the characteristic polynomial of A and the terms s_k are related by the following system of equations:

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & \cdots & & 0 \\ s_1 & 2 & 0 & \cdots & \cdots & & \vdots \\ s_2 & s_1 & 3 & \cdots & \cdots & & \vdots \\ s_3 & s_2 & s_1 & 4 & \cdots & & \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \\ & & & \ddots & \ddots & \ddots & \vdots \\ s_{n-1} & \cdots & & s_3 & s_2 & s_1 & n \end{bmatrix} \begin{bmatrix} c_{n-1} \\ c_{n-2} \\ c_{n-3} \\ \vdots \\ \vdots \\ c_1 \\ c_0 \end{bmatrix} = - \begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ \vdots \\ \vdots \\ s_{n-1} \\ s_n \end{bmatrix}. \quad (8.8)$$

The derivation of Eq. 8.8 is not as difficult as it may seem to be at first. It follows from the well-known **Newton's identities** relating the coefficients of an *arbitrary* polynomial to the sums of powers of the roots of the polynomial. A sketch of such a derivation is given in Exercises 8.26 and 8.27.

We have reduced the problem of computing the coefficients of $\phi(\lambda)$ to that of solving a triangular linear system of equations defined by $\{s_k \mid 1 \leq k \leq n-1\}$, where the terms s_k are the traces of the matrices A^ks. The traces can be computed trivially from the matrix powers $A^2, A^3, \ldots, A^{n-1}$, which can be computed fast in parallel. Moreover, we have already designed, in Section 8.2, a fast parallel algorithm to solve triangular systems. Therefore, we propose an algorithm whose main steps are these:

- *Step 1:* Compute $A^2, A^3, \ldots, A^{n-1}$.
- *Step 2:* Compute $s_k = tr(A^k)$, where $1 \leq k \leq n$.
- *Step 3:* Solve the triangular linear system of equations (Eq. 8.8) relating the coefficients of the characteristic polynomial to the terms s_k.

The following theorem gives the complexity of the algorithm for computing the characteristic polynomial.

Theorem 8.16: *Given an $n \times n$ matrix A, the coefficients of the characteristic polynomial of A can be determined in $O(\log^2 n)$ time, using a total of $O(nM(n))$ operations.*

Proof: The complexity bounds of each step can be estimated as follows. Step 1 can be handled by a prefix-sums algorithm. Hence, it requires $O(\log n)$ iterations, each of which involves a number of matrix multiplications. Therefore, the total time to execute step 1 is $O(\log^2 n)$. We clearly need $\Theta(n)$ matrix multiplications; hence, the total number of operations is $O(nM(n))$. Step 2 requires $O(\log n)$ time, using $O(n^2)$ operations; while step 3 can be done in $O(\log^2 n)$ time, using $O(M(n))$ operations. Therefore, the running time of the algorithm is $O(\log^2 n)$, and the total number of operations is $O(nM(n))$. □

Remark 8.2: The coefficient matrix of the linear system of equations defined by Eq. 8.8 is almost a Toeplitz matrix. It is possible to solve this system in $O(\log^2 n)$ time, using only $O(n \log n)$ operations. The details are left to Exercise 8.30. □

The results of Theorem 8.16 can be extended to cover three basic matrix computations.

Corollary 8.3: *Let A be an $n \times n$ matrix. Each of the following problems can be solved in $O(\log^2 n)$ time, using a total of $O(nM(n))$ operations:*

1. *Solve the linear system $\boldsymbol{Ax} = \boldsymbol{b}$, for any given vector $\boldsymbol{b}$.*
2. *Invert A, whenever A is nonsingular.*
3. *Compute* $\det(A)$.

□

Computation of the *LU* and *QR* Factorizations. Let A be an $n \times n$ nonsingular matrix. We wish to determine, whenever possible, a nonsingular lower triangular matrix L and a nonsingular upper triangular matrix U such that $A = LU$. This problem is referred to as the LU factorization of A.

Let n be a power of 2, and suppose that the LU factors of A exist. We then partition A, L and U into $\frac{n}{2} \times \frac{n}{2}$ blocks as follows:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{12} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix}.$$

This identity implies that $A_{11} = L_{11}U_{11}$, $A_{12} = L_{11}U_{12}$, $A_{21} = L_{12}U_{11}$, and $A_{22} = L_{12}U_{12} + L_{22}U_{22}$. Hence, an algorithm for computing the LU factors consists of the following steps:

- *Step 1:* Compute A_{11}^{-1}, if A_{11} is nonsingular; otherwise, declare that A does not have LU factors.

- *Step 2:* Compute $A_{11}^{-1}A_{12}$, $A_{21}A_{11}^{-1}$, and $A_{21}A_{11}^{-1}A_{12}$.
- *Step 3:* Recursively compute the LU factors of A_{11} (say, L_{11} and U_{11}), and the LU factors of $A_{22} - A_{21}A_{11}^{-1}A_{12}$ (say, L_{22} and U_{22}).
- *Step 4:* Set $L_{12} := (A_{21}A_{11}^{-1})L_{11}$ and $U_{12} := U_{11}(A_{11}^{-1}A_{12})$.

Let us verify the correctness of the algorithm. Since L and U are supposed to be nonsingular, L_{11} and U_{11} must be nonsingular, and hence A_{11} also is nonsingular. It is easy to check that A has an LU factorization if and only if A_{11} and $A_{22} - A_{21}A_{11}^{-1}A_{12}$ have LU factorizations.

The following theorem gives us the efficiency of our LU factorization algorithm.

Theorem 8.17: *Let A be an $n \times n$ nonsingular matrix that has an LU factorization. Then L and U can be determined in $O(\log^3 n)$ time, using a total of $O(nM(n))$ operations.*

Proof: Let $T(n)$ and $W(n)$ be the required time and number of operations, respectively. Then, $T(n)$ and $W(n)$ satisfy the following recurrence relations:

$$T(n) = T(n/2) + O(\log^2 n)$$
$$W(n) = I(n/2) + 5M(n/2) + 2W(n/2) + O(n^2),$$

where $I(n)$ is the number of operations required by our earlier algorithm to invert an $n \times n$ matrix in $O(\log^2 n)$ time. Clearly, $T(n) = O(\log^3 n)$ satisfies the first relation.

As for the recurrence relation of $W(n)$, we claim that $W(n) \leq 4I(n)$, under the reasonable assumption $I(n) \geq 4I(n/2)$. Assume that this bound holds for matrices of size $(n/2) \times (n/2)$. Then, $W(n) \leq I(n/2) + 5M(n/2) + 8I(n/2) + O(n^2) \leq 15I(n/2) \leq 4I(n)$. Therefore, the claimed complexity bounds follow. □

Related to LU factorization is the problem of QR factorization. Given an $n \times n$ nonsingular matrix A, we wish to find an **orthogonal matrix** Q (that is, Q satisfies $QQ^T = I$) such that $A = QR$, where R is a nonsingular upper triangular matrix. It turns out that the QR factors can be computed from the LU factors of A^TA (see Exercise 8.29). This fact leads us to the following theorem.

Theorem 8.18: *Let A be an $n \times n$ nonsingular matrix. Then, the QR factors of A can be computed in $O(\log^3 n)$ time using a total of $O(nM(n))$ operations.* □

PRAM Model: Since our matrix-multiplication algorithm is a basic building block, the CREW PRAM model can be used to achieve the stated bounds. □

8.8.2 ITERATIVE METHODS

Given that the arithmetic PRAM model performs various arithmetic operations exactly, the direct methods for handling matrix computations, described earlier in this chapter, generate exact solutions. In this section, we turn to **iterative methods** that will generate provably good approximations to our desired matrix computations. In reality, only finite-precision arithmetic can be used in matrix computations, resulting in round-off errors; hence, the solutions produced by the direct methods are *also* approximations to the exact solutions. However, such issues are outside the scope of this book. Iterative methods for matrix computations are widely used, especially for large systems, because these methods tend to be simpler, to be more robust to numerical errors, and to require less storage than direct methods. Here, we shall describe an iterative scheme for computing the inverse of a matrix.

The Approximation Notion. Let A be an $n \times n$ nonsingular matrix. We desire an efficient parallel method to compute a matrix B that can be viewed as a very good approximation to A^{-1}. The first question that comes to mind measure the closeness of the matrix B to A^{-1}. We shall examine the concept briefly; this notion has been the subject of extensive studies in numerical computations.

Let $x = [x_1, x_2, \ldots, x_n]$ be an n-dimensional vector with entries over a field. The **Euclidean norm** of the vector x is defined to be $\|x\| = (\sum_{i=1}^{n} x_i^2)^{\frac{1}{2}}$. This norm is usually denoted by $\|x\|_2$, but since it is the only norm we shall use, no confusion will arise if we drop the subscript. The *norm* of a matrix A is defined by

$$\|A\| = \max_{x \neq 0} \|Ax\| / \|x\|.$$

The following lemma lists some simple properties of the matrix norm. The first three properties reinforce the intuitive notion of a norm. Its proof is simple; it is left to Exercise 8.25.

Lemma 8.4: *Let A and B be two matrices of the same size. Then, the following properties hold:*

1. $\|A\| \geq 0$, *and* $\|A\| = 0$ *if and only if* $A = 0$.
2. $\|cA\| = |c| . \|A\|$, *for any scalar* c.
3. $\|A + B\| \leq \|A\| + \|B\|$.
4. $\|AB\| \leq \|A\| . \|B\|$. □

An $n \times n$ matrix B will be considered to be a good approximation to A^{-1} if the relative error $\|B - A^{-1}\| / \|A^{-1}\|$ is small. For our purposes, we will require the strong condition $\|B - A^{-1}\| / \|A^{-1}\| \leq q^{n^c}$, where $0 < q < 1$ and c is a positive integer.

An Iterative Scheme for Matrix Inversion. An iterative method consists of

1. A procedure to compute an initial approximation to the desired solution;
2. An iterative improvement scheme; and
3. A stopping criterion.

Given the $n \times n$ nonsingular matrix A, we let B_0 be an $n \times n$ matrix satisfying the condition $\|I - B_0A\| = q < 1$. We will later specify a simple way to determine B_0.

We define the **residual** of B_0 as $r(B_0) = I - B_0A$. Given our choice of B_0, we have $\|r(B_0)\| = q$. Our iterative improvement scheme consists of the following computation:

$$\text{set } B_k := (I + r(B_{k-1}))B_{k-1}.$$

Let us examine the closeness of B_k to A^{-1}. We start by bounding the residual corresponding to B_k, which is given by $r(B_k) = I - B_kA = I - B_{k-1}A - r(B_{k-1})B_{k-1}A = r(B_{k-1}) - r(B_{k-1})B_{k-1}A = [r(B_{k-1})]^2$. It follows that $r(B_k) = [r(B_0)]^{2^k}$. Therefore, $\|r(B_k)\| \leq \|r(B_0)\|^{2^k} = q^{2^k}$. On the other hand, $\|B_k - A^{-1}\| = \|(B_kA - I)A^{-1}\| \leq \|B_kA - I\|.\|A^{-1}\| \leq \|r(B_k)\|.\|A^{-1}\|$, and thus

$$\frac{\|B_k - A^{-1}\|}{\|A^{-1}\|} \leq q^{2^k}.$$

If we set $k = c \lceil \log n \rceil$, we get our desired approximation to A^{-1}.

It remains for us to show how to determine the initial approximation B_0. It turns out that the simple choice of $B_0 = \alpha A^T$, where $\alpha = 1/tr(A^TA)$, will satisfy the condition $\|r(B_0)\| = q < 1$.

In summary, we have proved the following theorem.

Theorem 8.19: *Let A be an $n \times n$ invertible matrix, and let $B_0 = \alpha A^T$, where $\alpha = 1/tr(A^TA)$. Then, $\|r(B_0)\| = \|I - B_0A\| = q < 1$, and the matrix B_k generated by the iterative method $B_k = (I + r(B_{k-1}))B_{k-1}$ satisfies $\|B_k - A^{-1}\|/\|A^{-1}\| \leq q^{2^k}$. Therefore, a good approximation to A^{-1} can be found in $O(\log^2 n)$ time, using a total of $O(M(n) \log n)$ operations.*

Proof: The correctness proof follows from the discussions preceding the statement of the theorem. The error bound implies that $O(\log n)$ iterations are sufficient to guarantee an approximation satisfying our condition. Each such iteration can be done in $O(\log n)$ time, using $O(M(n))$ operations. □

PRAM Model: The basic computational step needed to implement the iterative algorithm described is matrix multiplication. Hence, a CREW PRAM model is needed to achieve the bounds stated in the theorem. □

8.9 *Dense Structured Matrices

As we saw in Section 8.8, the direct methods require an excessive number of operations to carry out basic matrix computations fast. We will now explore the possibility of reducing the amount of work in the case where the dense matrices possess certain regular structures, as do Toeplitz matrices, for example. Informally, we refer to such matrices as **structured.** These types of matrices are used in many applications; hence, related computations are interesting in their own right.

8.9.1 EXAMPLES OF NON-TOEPLITZ STRUCTURED MATRICES

To get a feel for what we mean by *structured matrices*, we begin by examining a couple of non-Toeplitz matrices, before continuing our examination of Toeplitz matrices.

An $m \times n$ **Vandermonde matrix** $V = (v_{ij})$ corresponds to a vector $\boldsymbol{p} = [p_0, \ldots, p_{m-1}]$, where $v_{ij} = p_i^j$, $0 \le i \le m - 1$ and $0 \le j \le n - 1$; that is,

$$V = \begin{bmatrix} 1 & p_0 & p_0^2 & \cdots & p_0^{n-1} \\ 1 & p_1 & p_1^2 & \cdots & p_1^{n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & p_{m-1} & p_{m-1}^2 & \cdots & p_{m-1}^{n-1} \end{bmatrix}.$$

The matrix-vector product $\boldsymbol{Va}$, where $\boldsymbol{a} = [a_0, \ldots, a_{n-1}]$, corresponds to the evaluation of the polynomial $Q(x) = \sum_{i=0}^{n-1} a_i x^i$ at the m points p_0 $p_1, \ldots, p_{m-1}$. On the other hand, solving the linear system of equations $\boldsymbol{Va} = \boldsymbol{b}$ (and $m = n$) corresponds to polynomial interpolation.

Another non-Toeplitz structured matrix is called the **Sylvester** or **resultant matrix** $S(p, q)$ of two polynomials $p(x)$ and $q(x)$. If $p(x) = \sum_{i=0}^{n} u_i x^i$ and $q(x) = \sum_{i=0}^{m} v_i x^i$, then $S(p, q)$ is the $(m + n) \times (m + n)$ matrix defined as follows:

$$S(p,q) = \begin{bmatrix} u_n & & & & v_m & & & \\ u_{n-1} & \ddots & & & v_{m-1} & \ddots & & \\ \vdots & \ddots & u_n & & \vdots & \ddots & v_m & \\ & & u_{n-1} & & & & v_{m-1} & \\ & & \vdots & & & & \vdots & \\ u_1 & & & & v_1 & & & \\ u_0 & \ddots & & & v_0 & \ddots & & \\ & \ddots & u_1 & & & \ddots & v_1 & \\ & & u_0 & & & & v_0 & \end{bmatrix}.$$

That is, $S(p,q) = [T_1, T_2]$, where T_1 is an $(n + m) \times m$ Toeplitz matrix whose first column is $[u_n, u_{n-1}, \ldots, u_0, 0, \ldots, 0]^T$ and whose first row is $[u_n, 0, \ldots, 0]$, and T_2 is an $(n + m) \times n$ Toeplitz matrix whose first column is $[v_m, v_{m-1}, \ldots, v_0, 0, \ldots, 0]^T$ and whose first row is $[v_m, 0, \ldots, 0]$. For example, the Sylvester matrix corresponding to $p(x) = u_0 + u_1x + u_2x^2$ and $q(x) = v_0 + v_1x + v_2x^2 + v_3x^3$ is

$$\begin{bmatrix} u_2 & 0 & 0 & v_3 & 0 \\ u_1 & u_2 & 0 & v_2 & v_3 \\ u_0 & u_1 & u_2 & v_1 & v_2 \\ 0 & u_0 & u_1 & v_0 & v_1 \\ 0 & 0 & u_0 & 0 & v_0 \end{bmatrix}.$$

The determinant of $S(p,q)$ is called the **resultant** of the two polynomials $p(x)$ and $q(x)$. An important application of the resultant is indicated in the following well-known theorem of algebra.

Theorem 8.20: *Let* $p(x) = \sum_{i=0}^{n} u_i x^i$ *and* $q(x) = \sum_{i=0}^{m} v_i x^i$ *be two polynomials over a field* $\mathcal{F}$ *such that* $u_n \neq 0$ *and* $v_m \neq 0$. *Then, the resultant is equal to* 0 *if and only if* $p(x)$ *and* $q(x)$ *have a common factor of positive degree in* $\mathcal{F}[x]$. *In particular, the two polynomials have a common root (which may not be in* $\mathcal{F}$*) if and only if the resultant is equal to* 0. □

Another application of the Sylvester matrix will occur in the algorithm to compute the *greatest common divisor*, described in Section 8.9.3. The greatest-common-divisor algorithm requires the solution of a linear system whose coefficient matrix is a submatrix of the Sylvester matrix, which we call a **subresultant matrix.**

In the meantime, we return to the problem of computing matrix inversion iteratively; we apply the results to Toeplitz matrices in particular.

8.9.2 NEWTON'S ITERATION FOR MATRIX INVERSION

Our starting point will be to adapt the iterative algorithm for matrix inversion that we discussed in Section 8.8.2. We shall replace the notion of a norm by that of modular polynomial arithmetic.

Let A be an arbitrary $n \times n$ matrix and let λ be an indeterminate. Since $(I - \lambda A)(I + \lambda A + \lambda^2 A^2 \cdots + \lambda^k A^k) = I - \lambda^{k+1} A^{k+1}$, for any positive integer k, we have that $(I - \lambda A)(I + \lambda A + \cdots + \lambda^k A^k) = I \bmod \lambda^{k+1}$. That is, $(I - \lambda A)^{-1} \bmod \lambda^{k+1} = I + \lambda A \cdots + \lambda^k A^k$.

We now state the basic algorithm for computing $(I - \lambda A)^{-1}$. (Compare this algorithm with the iterative scheme in Section 8.8.2 for computing the inverse.)

ALGORITHM 8.5
(Newton's Iteration)

Input: *An arbitrary* $n \times n$ *matrix* A, *where* $n + 1$ *is a power of* 2.
Output: *The matrix* $(I - \lambda A)^{-1} \bmod \lambda^{n+1}$, *where* λ *is an indeterminate.*

begin
1. *Set* $X_0 := I, B := I - \lambda A$
2. **for** $i = 1$ **to** $\log (n + 1)$ **do**
   Set $X_i := (2I - X_{i-1}B)X_{i-1}$

end

Lemma 8.5: *The matrix* X_i *generated during the* i*th iteration of Newton's iteration (Algorithm 8.5) is given by* $X_i = (I - \lambda A)^{-1} \bmod \lambda^{2^i}$.

Proof: We prove by induction on i that the matrix X_i generated at the ith iteration of the **for** loop of step 2 satisfies $X_i = (I - \lambda A)^{-1} \bmod \lambda^{2^i}$. Since $(I - \lambda A)^{-1} \bmod \lambda^{2^i} = I + \lambda A \cdots + \lambda^{2^i-1}A^{2^i-1}$, it is sufficient to prove that $X_i = I + \lambda A + \cdots + \lambda^{2^i-1}A^{2^i-1}$.

The base case is where $i = 1$ and hence $A_1 = (2I - X_0B)X_0 = I + \lambda A$, and the induction hypothesis holds in this case.

Assume that the induction hypothesis holds for X_i. We show that it also holds for X_{i+1}.

The matrix X_{i+1} is given by $X_{i+1} = (2I - X_i(I - \lambda A))X_i$. By the induction hypothesis, we have $X_i = I + \lambda A + \cdots + \lambda^{2^i-1}A^{2^i-1}$; hence, $X_i(I - \lambda A) = I - \lambda^{2^i}A^{2^i}$. It follows that $X_{i+1} = (I + \lambda^{2^i}A^{2^i})(I + \lambda A + \cdots + \lambda^{2^i-1}A^{2^i-1}) = I + \lambda A + \cdots + \lambda^{2^{i+1}-1}A^{2^{i+1}-1}$; hence, the lemma follows for all values of i. □

We let $l = \log (n + 1)$, assuming as before that $n + 1$ is a power of 2. Lemma 8.5 implies that $X_l = \sum_{i=0}^{n} \lambda^i A^i$. Thus, the trace of the matrix X_l is given by $tr(X_l) = \sum_{i=0}^{n} \lambda^i\, tr(A^i)$. Therefore, the traces of all the powers of the matrix A can be deduced immediately from X_l. Once we have $tr(A^i)$, where $1 \le i \le n$, we can compute the coefficients of the characteristic polynomial of A efficiently (see Remark 8.2 in Section 8.8).

Let us determine the computational requirements of Algorithm 8.5. The algorithm consists of $O(\log n)$ iterations, and each iteration requires two $n \times n$ matrix multiplications. However, these matrices are polynomial matrices; hence, each multiplication of two entries of these matrices is between two polynomials in λ. It is easy to verify that Algorithm 8.5 does not really represent a gain in efficiency over the previous algorithms of Section 8.8. However, in the case of Toeplitz matrices, to which we next turn, we shall indeed be able to exploit their special structure and to reduce the number of operations substantially.

8.9.3 NEWTON'S ITERATION FOR TOEPLITZ MATRIX INVERSION

From our earlier discussion of Toeplitz matrices (Section 8.5), recall that the entries on each diagonal of a Toeplitz matrix $\boldsymbol{T}$ are equal, and that the matrix-vector product $\boldsymbol{Tx}$ can be reduced to a convolution of two vectors (and hence to the product of two polynomials). In particular, the multiplication of an $n \times n$ Toeplitz matrix $\boldsymbol{T}$ by an n-dimensional vector can be performed in $O(\log n)$ time, using a total of $O(n \log n)$ operations (Lemma 8.1).

The key observation that will allow the development of a more efficient algorithm derives from the following theorem.

Theorem 8.21: *Let $\boldsymbol{T}^{-1}$ be the inverse of an $n \times n$ Toeplitz matrix $\boldsymbol{T}$, and let $\boldsymbol{u} = [u_1, \ldots, u_n]^T$ and $\boldsymbol{v} = [v_1, \ldots, v_n]^T$ be the two vectors representing the first and the last columns, respectively, of $\boldsymbol{T}^{-1}$. Let $L(\boldsymbol{u})$ denote the lower triangular Toeplitz matrix whose first column is $\boldsymbol{u}$, and let $U(\boldsymbol{v})$ be the upper triangular Toeplitz matrix whose first row is $\boldsymbol{v}$. Then*

$$u_1 \boldsymbol{T}^{-1} = \boldsymbol{L(u)U}(\boldsymbol{v}^{(r)}) - \boldsymbol{L}(\boldsymbol{v}^{(s)})\boldsymbol{U}(\boldsymbol{u}^{(t)}),$$

where $\boldsymbol{v}^{(r)} = [v_n, v_{n-1}, \ldots, v_1]^T$, $\boldsymbol{v}^{(s)} = [0, v_1, \ldots, v_{n-1}]^T$, and $\boldsymbol{u}^{(t)} = [0, u_n, \ldots, u_2]^T$. □

The proof of this theorem can be found in several of the references given at the end of this chapter (see bibliographic notes).

EXAMPLE 8.20:

Let $n = 3$. Then

$$u_1 \boldsymbol{T}^{-1} = \begin{bmatrix} u_1 & 0 & 0 \\ u_2 & u_1 & 0 \\ u_3 & u_2 & u_1 \end{bmatrix} \begin{bmatrix} v_3 & v_2 & v_1 \\ 0 & v_3 & v_2 \\ 0 & 0 & v_3 \end{bmatrix} - \begin{bmatrix} 0 & 0 & 0 \\ v_1 & 0 & 0 \\ v_2 & v_1 & 0 \end{bmatrix} \begin{bmatrix} 0 & u_3 & u_2 \\ 0 & 0 & u_3 \\ 0 & 0 & 0 \end{bmatrix}.$$ □

Let us now revisit the basic Newton iteration $\boldsymbol{X}_i := (2\boldsymbol{I} - \boldsymbol{X}_{i-1}\boldsymbol{B})\boldsymbol{X}_{i-1}$. Since $\boldsymbol{X}_i = (\boldsymbol{I} - \lambda \boldsymbol{A})^{-1} \bmod \lambda^{2^i}$, Theorem 8.21 implies that $\boldsymbol{X}_i$ is uniquely determined by its first and its last columns. Suppose the first column $\boldsymbol{u}_{i-1}$ and last column $\boldsymbol{v}_{i-1}$ of $\boldsymbol{X}_{i-1}$ are given. Then, using the theorem, we can see that the first column of $\boldsymbol{u}_i$ of $\boldsymbol{X}_i$ is given by

$$\boldsymbol{u}_i = (2\boldsymbol{I} - \boldsymbol{X}_{i-1}\boldsymbol{B})\boldsymbol{u}_{i-1};$$

that is,

$$\boldsymbol{u}_i = 2\boldsymbol{u}_{i-1} - c\,[\boldsymbol{L}(\boldsymbol{u}_{i-1})\boldsymbol{U}(\boldsymbol{v}_{i-1}^{(r)}) - \boldsymbol{L}(\boldsymbol{v}_{i-1}^{(s)})\,\boldsymbol{U}(\boldsymbol{u}_{i-1}^{(t)})]\boldsymbol{B}\boldsymbol{u}_{i-1},$$

where c is equal to the inverse of the first entry of $\boldsymbol{u}_{i-1}$. Lemma 8.5 can be used to show that the $(1, 1)$ entry of $\boldsymbol{X}_i$ has the form $1 + a_1\lambda + a_2\lambda^2 + \ldots + a_{2^{i-1}}\lambda^{2^{i-1}}$, and hence its inverse c will always exist.

We conclude that we can obtain the column $\boldsymbol{u}_i$ from $\boldsymbol{u}_{i-1}$ and $\boldsymbol{v}_{i-1}$ by applying several matrix-vector products, where each matrix is a Toeplitz matrix. We have already seen, in Section 8.5, that such a computation can be reduced to polynomial multiplication (or convolution). Since the elements involved are polynomials in λ, such a matrix-vector product is now reduced to polynomial multiplication where each polynomial has two variables.

Polynomial multiplication with several variables can be handled by iterating the one-variable case. For the two-variable case, we obtain an algorithm running in $O(\log n)$ time and using a total of $O(n^2 \log n)$ operations, with the assumption that our field supports the FFT algorithm. The details are left to Exercise 8.13.

Obviously, a similar procedure applies to the problem of computing $\boldsymbol{v}_i$.

This discussion suggests that each iteration of Algorithm 8.5 can be done in $O(\log n)$ time, using a total of $O(n^2 \log n)$ operations, whenever the input matrix is a Toeplitz matrix. Therefore, we have the following theorem.

Theorem 8.22: *For an $n \times n$ Toeplitz matrix $\boldsymbol{T}$, we can compute $(\boldsymbol{I} - \lambda \boldsymbol{T})^{-1} \bmod \lambda^{n+1}$ in $O(\log^2 n)$ time, using a total of $O(n^2 \log^2 n)$ operations.* □

Corollary 8.4: *If $\boldsymbol{T}$ is an $n \times n$ Toeplitz matrix, then the quantities $s_i = tr(\boldsymbol{T}^i)$, where $1 \leq i \leq n - 1$, can be computed in $O(\log^2 n)$ time, using $O(n^2 \log^2 n)$ operations.* □

Corollary 8.5: *The characteristic polynomial of an $n \times n$ Toeplitz matrix $\boldsymbol{T}$ can be computed in $O(\log^2 n)$ time, using $O(n^2 \log^2 n)$ operations. Therefore, computation of the determinant of a Toeplitz matrix can also be carried out within these bounds.*

Proof: As we have seen before, the coefficients of the characteristic polynomial of $\boldsymbol{T}$ can be obtained from $\{s_i = tr(\boldsymbol{T}^i)\}_{1 \leq i \leq n-1}$ in $O(\log^2 n)$ time, using a total of $O(n \log n)$ operations (see Remark 8.2). □

Corollary 8.6: *Solving a linear system of equations whose coefficient matrix is an $n \times n$ Toeplitz matrix can be done in $O(\log^2 n)$ time, using a total of $O(n^2 \log^2 n)$ operations.*

Proof: By Corollary 8.5, we can compute the characteristic polynomial of the coefficient matrix in $O(\log^2 n)$ time, using a total of $O(n^2 \log^2 n)$ operations. We have already seen, in Lemma 8.3, how we can solve a linear system from the coefficients of the characteristic polynomial. We can use the same method

here, in combination with Theorem 8.22 and the fact that $(I - \lambda T)^{-1} \bmod \lambda^{n+1}$ is uniquely determined by its first and last columns, to solve the Toeplitz linear system of equations in $O(\log^2 n)$, using a total of $O(n^2 \log^2 n)$ operations. □

These results for Toeplitz matrices apply to resultant matrices as well. As a matter of fact, the same strategy can be used to handle the class of Toeplitz-like matrices, which in particular contains resultant and subresultant matrices. Hence, we shall assume in what follows that a solution of a linear system whose coefficient matrix is a subresultant can be determined in $O(\log^2 n)$ time, using $O(n^2 \log^2 n)$ operations.

8.9.4 DETERMINATION OF THE POLYNOMIAL GREATEST COMMON DIVISOR

One application of the Sylvester matrix, introduced in Section 8.9.1 is computing the **greatest common divisor (gcd)** of two polynomials. The gcd of the two polynomials $u(x) = \sum_{i=0}^{n} u_i x^i$ and $v(x) = \sum_{i=0}^{m} v_i x^i$ over a field $\mathcal{F}$ is a polynomial $\delta(x) \in \mathcal{F}[x]$ such that the following two conditions are true: (1) $\delta(x)$ divides both $u(x)$ and $v(x)$, and (2) every divisor of $u(x)$ and $v(x)$ divides $\delta(x)$. In other words, the gcd of $u(x)$ and $v(x)$ is a maximum-degree polynomial that divides both $u(x)$ and $v(x)$. To make the gcd of $u(x)$ and $v(x)$ unique, we can require that $\delta(x)$ be *monic*; that is, the coefficient of the highest degree term of $\delta(x)$ is equal to 1.

EXAMPLE 8.21:

Let $u(x) = x^3 - 2x^2 + 2x - 1$ and $v(x) = x^4 - x^3 + 2x^2 - x + 1$ be two polynomials over the field of real numbers. Then, $u(x) = (x - 1)(x^2 - x + 1)$ and $v(x) = (x^2 + 1)(x^2 - x + 1)$. Hence, $\delta(x) = x^2 - x + 1$ is the gcd of the two polynomials $u(x)$ and $v(x)$. □

Extended Euclid's Algorithm. We start by outlining **Euclid's algorithm** for computing the gcd of the two polynomials $u(x) = \sum_{i=0}^{n} u_i x^i$ and $v(x) = \sum_{i=0}^{m} v_i x^i$. Without loss of generality, we assume that $m \leq n$.

We set $u_0(x) = u(x)$ and $u_1(x) = v(x)$, and start by dividing $u_0(x)$ by $u_1(x)$ to obtain

$$u_0(x) = q_1(x)u_1(x) + u_2(x),$$

where $q_1(x)$ is the quotient and $u_2(x)$ is the remainder. Hence, $\deg(u_2(x)) < \deg(u_1(x))$. Any divisor of $u_0(x)$ and $u_1(x)$ is a divisor of $u_1(x)$ and $u_2(x)$, and, conversely, any divisor of $u_1(x)$ and $u_2(x)$ is a divisor of $u_0(x)$ and $u_1(x)$. Therefore, $gcd(u_0(x), u_1(x)) = gcd(u_1(x), u_2(x))$.

We now divide $u_1(x)$ by $u_2(x)$; we obtain

$$u_1(x) = q_2(x)u_2(x) + u_3(x),$$

where $\deg(u_3(x)) < \deg(u_2(x))$. As before, $gcd(u_2(x), u_3(x)) = gcd(u_1(x), u_2(x)) = gcd(u(x), v(x))$, and hence the process can be continued until we have

$$u_{k-1}(x) = q_k(x)u_k(x);$$

that is, the remainder $u_{k+1}(x)$ is equal to 0. Since the degrees of the successive remainders are strictly decreasing, we will reach this stage after at most m iterations. We now conclude that $\delta(x) = gcd(u(x), v(x)) = u_k(x)$ since $gcd(u_{k-1}(x), u_k(x)) = u_k(x)$, and Euclid's algorithm terminates.

The best-known sequential algorithm for computing the gcd of two polynomials is based on a divide-and-conquer implementation of Euclid's algorithm that requires $O(n \log^2 n)$ operations. Starting from Euclid's algorithm, we shall derive a different algorithm that is more suitable for parallel processing.

Using the **remainder sequence** $u_{k-1}(x), u_{k-2}(x), \ldots, u_0(x)$ described earlier, we can express the gcd $u_k(x)$ as follows: $u_k(x) = p(x)u(x) + q(x)v(x)$, for some polynomials $p(x)$ and $q(x)$. In fact, the **extended Euclidean scheme** computes the gcd as well as the two polynomials $p(x)$ and $q(x)$ as follows.

Let $u_0(x) = u(x)$, $u_1(x) = v(x)$, $v_0(x) = 0$, $v_1(x) = 1$, $w_0(x) = 1$, and $w_1(x) = 0$. Then, for $i = 1, 2, \ldots, k$, where $u_{k+1}(x) = 0$, we compute the following:

$$\begin{aligned} u_{i+1}(x) &= -q_i(x)u_i(x) + u_{i-1}(x) \\ v_{i+1}(x) &= -q_i(x)v_i(x) + v_{i-1}(x) \\ w_{i+1}(x) &= -q_i(x)w_i(x) + w_{i-1}(x), \end{aligned}$$

where $\deg u_{i+1}(x) < \deg u_i(x)$. The computation of the $\{u_i\}$ sequence is identical to the one described before. Our interest in the two sequences $\{v_i\}$ and $\{w_i\}$ is due to the following identity, which can be easily shown by induction on i:

$$w_i(x)u(x) + v_i(x)v(x) = u_i(x).$$

Therefore, $gcd(u(x), v(x)) = u_k(x) = w_k(x)u(x) + v_k(x)v(x)$.

In summary, we have shown that the greatest common divisor $\delta(x)$ of two polynomials $u(x)$ and $v(x)$ can be expressed as $\delta(x) = p(x)u(x) + q(x)v(x)$, for some polynomials $p(x)$ and $q(x)$.

On the other hand, any polynomial $s(x)$ of the form $s(x) = \alpha(x)u(x) + \beta(x)v(x)$, for any two polynomials $\alpha(x)$ and $\beta(x)$, is divisible by $\delta(x)$ since $\delta(x)$ divides both $u(x)$ and $v(x)$. We conclude that $\delta(x)$ is the *smallest-degree polynomial that can be written in the form* $\delta(x) = p(x)u(x) + q(x)v(x)$, for some polynomials $p(x)$ and $q(x)$. This fact can be expressed in more abstract terms as follows.

Remark 8.3: Let $\mathcal{F}[x]$ be the ring of polynomials over the field $\mathcal{F}$. Then, the ideal generated by any two polynomials $u(x)$ and $v(x)$ in $\mathcal{F}[x]$ consists of all the elements that are multiples of the greatest common divisor $\delta(x)$. □

A Fast Parallel Algorithm. Our parallel algorithm will identify the two polynomials $p(x)$ and $q(x)$ defining $\delta(x)$ as follows.

Let the degree of $\delta(x)$ be k. Then, using the remainder sequence generated by the extended Euclidean scheme, it is easy to show by induction that $\deg(p(x)) \leq m - k - 1$ and $\deg(q(x)) \leq n - k - 1$ (Exercise 8.38). Hence, suppose that $p(x) = \sum_{i=0}^{m-k-1} p_i x^i$ and $q(x) = \sum_{i=0}^{n-k-1} q_i x^i$, where the p_i's and the q_i's are to be determined.

As we saw in Section 8.5.1, each polynomial product can be expressed as a Toeplitz matrix-vector product. In particular, the coefficients of $p(x)u(x)$ are given by

$$\begin{bmatrix} u_n & & & & & \\ u_{n-1} & u_n & & & & \\ \vdots & u_{n-1} & u_n & & & \\ u_1 & \vdots & \ddots & \ddots & & \\ u_0 & u_1 & & \ddots & u_n & \\ & u_0 & u_1 & & u_{n-1} & \\ & & \ddots & \ddots & \vdots & \\ & & & \ddots & u_1 & \\ & & & & u_0 & \end{bmatrix} \begin{bmatrix} p_{m-k-1} \\ \vdots \\ \vdots \\ p_1 \\ p_0 \end{bmatrix}, \tag{8.9}$$

where the coefficient matrix is Toeplitz of size $(n + m - k) \times (m - k)$. A similar matrix-vector product holds for the polynomial multiplication $q(x)v(x)$.

Since the gcd of the two polynomials $u(x)$ and $v(x)$ satisfies $\delta(x) = p(x)u(x) + q(x)v(x)$, its coefficients are given by the following product:

$$\begin{bmatrix} u_n & & & & v_m & & & \\ u_{n-1} & u_n & & & v_{m-1} & v_m & & \\ \vdots & u_{n-1} & u_n & & \vdots & v_{m-1} & v_m & \\ & \ddots & \ddots & \ddots & & \ddots & \ddots & \ddots \\ \vdots & & \ddots & \ddots & \vdots & & \ddots & \ddots \\ u_0 & & & u_n & v_0 & & & v_m \\ & u_0 & & u_{n-1} & & v_0 & & v_{m-1} \\ & & \ddots & \vdots & & & \ddots & \vdots \\ & & & \vdots & & & & \vdots \\ & & & u_0 & & & & v_0 \end{bmatrix} \begin{bmatrix} p_{m-k-1} \\ \vdots \\ \vdots \\ p_1 \\ p_0 \\ q_{n-k-1} \\ \vdots \\ \vdots \\ q_1 \\ q_0 \end{bmatrix}, \tag{8.10}$$

where the coefficient matrix is of size $(n + m - k) \times (n + m - 2k)$.

The coefficient of x^j in the polynomial $p(x)u(x) + q(x)v(x)$ must be 0 for all $j > k$, and the coefficient of x^k must be equal to 1 (which ensures that $\delta(x)$ is monic of degree k). We therefore obtain the following linear system of equations by removing the last k rows of the coefficient matrix in Eq. 8.10:

$$\begin{bmatrix} u_n & & & & v_m & & & \\ u_{n-1} & u_n & & & v_{m-1} & v_m & & \\ \vdots & u_{n-1} & u_n & & \vdots & v_{m-1} & v_m & \\ & \ddots & \ddots & \ddots & & \ddots & \ddots & \ddots \\ u_0 & & \ddots & u_n & & & \ddots & v_m \\ & u_0 & & u_{n-1} & v_0 & & & v_{m-1} \\ & & \ddots & \vdots & & \ddots & & \vdots \\ & & u_0 & u_k & & & v_0 & v_k \end{bmatrix} \begin{bmatrix} p_{m-k-1} \\ \vdots \\ \vdots \\ p_1 \\ p_0 \\ q_{n-k-1} \\ \vdots \\ \vdots \\ q_1 \\ q_0 \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ \vdots \\ 0 \\ 1 \end{bmatrix}. \quad (8.11)$$

For example, for $n = 4, m = 5$ and $k = 1$, we get

$$\begin{bmatrix} u_4 & 0 & 0 & 0 & v_5 & 0 & 0 \\ u_3 & u_4 & 0 & 0 & v_4 & v_5 & 0 \\ u_2 & u_3 & u_4 & 0 & v_3 & v_4 & v_5 \\ u_1 & u_2 & u_3 & u_4 & v_2 & v_3 & v_4 \\ u_0 & u_1 & u_2 & u_3 & v_1 & v_2 & v_3 \\ 0 & u_0 & u_1 & u_2 & v_0 & v_1 & v_2 \\ 0 & 0 & u_0 & u_1 & 0 & v_0 & v_1 \end{bmatrix} \begin{bmatrix} p_3 \\ p_2 \\ p_1 \\ p_0 \\ q_2 \\ q_1 \\ q_0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

Note that the coefficient matrix of the system in Eq. 8.11 is of size $(n + m - 2k) \times (n + m - 2k)$, and is a submatrix of the Sylvester matrix associated with the two polynomials $u(x)$ and $v(x)$. As we indicated near the beginning of this section, such a matrix is referred to as a *subresultant*.

EXAMPLE 8.22:

Consider the two polynomials $u(x) = x^3 - 2x^2 + 2x - 1$ and $v(x) = x^4 - x^3 + 2x^2 - x + 1$ whose gcd polynomial (of degree 2) was found in Example 8.21. Let $p(x) = p_0 + p_1x$ and $q(x) = q_0$. Then, $p(x)u(x) + q(x)v(x) = (p_1 + q_0)x^4 + (p_0 - 2p_1 - q_0)x^3 + (-2p_0 + 2p_1 + 2q_0)x^2 + (2p_0 - p_1 - q_0)x + q_0 - p_0$. Thus, we must have $p_1 + q_0 = p_0 - 2p_1 - q_0 = 0$ and $-2p_0 + 2p_1 + 2q_0 = 1$. In matrix form, this result would be

$$\begin{bmatrix} 1 & 0 & 1 \\ -2 & 1 & -1 \\ 2 & -2 & 2 \end{bmatrix} \begin{bmatrix} p_1 \\ p_0 \\ q_0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

Solving this linear system of equations, we obtain $p_0 = p_1 = -\frac{1}{2}$ and $q_0 = \frac{1}{2}$. Substituting these values into $p(x)u(x) + q(x)v(x)$, we get $x^2 - x + 1$, which is indeed the gcd of $u(x)$ and $v(x)$. □

Since the coefficient matrix of the linear system defined by Eq. 8.11 consists of two Toeplitz blocks, it can be shown that the techniques used to handle Toeplitz computations can be extended to this case as well. Hence, if the field $\mathcal{F}$ supports the FFT algorithm and is of characteristic zero, the linear system defined by Eq. 8.11 can be solved in $O(\log^2 n)$ time, using a total of $O(n^2 \log^2 n)$ operations (Corollary 8.6). Therefore, the coefficients of $\delta(x)$ can be obtained in $O(\log^2 n)$ time, using a total of $O(n^2 \log^2 n)$ operations.

The algorithm described here is not yet complete, since we do not know the degree k of the gcd of the two input polynomials. However, the following fact will allow us to perform binary search to determine the degree k.

Lemma 8.6: *Let $S(u, v)$ be the Sylvester matrix of the two polynomials $u(x)$ and $v(x)$ of degrees n and m, respectively, such that $m \leq n$. Given any integer t such that $0 \leq t \leq m$, then $t \geq \deg(gcd(u(x), v(x)))$ if and only if the linear system (Eq. 8.11) corresponding to $S(u, v)$, where the last t rows are removed, has a solution.* □

We can obtain the proof of this lemma by multiplying the equation $\delta(x) = p(x)u(x) + q(x)v(x)$ by x^{t-k} and deducing its matrix counterpart. The simple details are left to Exercise 8.38.

We have shown that we can use the binary search method to find the desired k, and then can determine the corresponding solution. We thus have the following theorem.

Theorem 8.23: *The gcd of two polynomials of degrees n and m, respectively, can be found in $O(\log^3(n + m))$ time, using a total of $O((m + n)^2 \log^3(n + m))$ operations.* □

PRAM Model: It is easy to check that all the algorithms in this section can be implemented on the CREW PRAM within the stated bounds. □

8.10 Summary

We considered, in this chapter, several fundamental operations on matrices and polynomials. We were able to develop fast parallel algorithms for all the

problems considered. In particular, the total number of operations used by each of our algorithms for handling polynomial computations matches that of the best known sequential algorithms, except for the gcd computation. As for matrix computations, most of our algorithms require considerably more operations than do the corresponding sequential algorithms. Tables 8.1 and 8.2 summarize the complexity bounds of the important algorithms covered in this chapter. Table 8.1 covers matrix computations; Table 8.2 covers polynomial computations.

TABLE 8.1
MATRIX COMPUTATIONS.

	Algorithm	Section	$T(n)$	$W(n)$	PRAM Model
8.1	First-Order Linear Recurrence	8.1.2	$O(\log n)$	$O(n)$	EREW
	Krylov Matrix	8.1.3	$O(\log^2 n)$	$O(M(m) \log n)$	CREW
	Triangular Linear Systems	8.2.1	$O(\log^2 n)$	$O(M(n))$	CREW
	Banded Triangular Linear Systems	8.2.2	$O(\log n \log m)$	$O\left(\frac{nM(m)}{m}\right)$	CREW
8.2	Fast Fourier Transform	8.3.2	$O(\log n)$	$O(n \log n)$	EREW
	Toeplitz Matrix-Vector Product	8.5.1	$O(\log n)$	$O(n \log n)$	EREW
	Triangular Toeplitz Matrix Inversion	8.5.2	$O(\log^2 n)$	$O(n \log n)$	EREW
	Circulant Matrix Inversion	8.5.3	$O(\log n)$	$O(n \log n)$	EREW
	Characteristic Polynomial	8.8.1	$O(\log^2 n)$	$O(nM(n))$	CREW
	Linear Systems	8.8.1	$O(\log^2 n)$	$O(nM(n))$	CREW
	Matrix Inversion	8.8.1	$O(\log^2 n)$	$O(nM(n))$	CREW
	Determinant	8.8.1	$O(\log^2 n)$	$O(nM(n))$	CREW
	LU-Factorization	8.8.1	$O(\log^3 n)$	$O(nM(n))$	CREW
	QR-Factorization	8.8.1	$O(\log^3 n)$	$O(nM(n))$	CREW
	Interative Matrix Inversion	8.8.2	$O(\log^2 n)$	$O(M(n) \log n)$	CREW
	Characteristic Polynomial of a Toeplitz Matrix	8.9.3	$O(\log^2 n)$	$O(n^2 \log^2 n)$	CREW
	Toeplitz Linear System	8.9.3	$O(\log^2 n)$	$O(n^2 \log^2 n)$	CREW

$M(n)$: best known bound for the number of arithmetic operations required for $n \times n$ matrix multiplication.

TABLE 8.2
POLYNOMIAL COMPUTATIONS.

	Algorithm	Section	$T(n)$	$W(n)$	PRAM Model
	Horner's Algorithm	8.1.1 and 8.1.2	$O(\log n)$	$O(n)$	EREW
	Polynomial Evaluation and Interpolation at Roots of Unity	8.4.1	$O(\log n)$	$O(n \log n)$	EREW
	Polynomial Multiplication	8.4.2	$O(\log n)$	$O(n \log n)$	EREW
	Polynomial Division	8.6	$O(\log^2 n)$	$O(n \log n)$	EREW
8.3	Polynomial Evaluation at n Distinct Points	8.7.1	$O(\log^3 n)$	$O(n \log^2 n)$	EREW
8.4	Polynomial Interpolation	8.7.2	$O(\log^3 n)$	$O(n \log^2 n)$	EREW
	Greatest Common Divisor	8.9.4	$O(\log^3 n)$	$O(n^2 \log^3 n)$	CREW

Exercises

8.1. Develop a nonrecursive version of Algorithm 8.1 (first-order linear recurrence). Outline a solution to the corresponding processor-allocation problem.

8.2. Consider the matrix linear recurrence

$$\boldsymbol{Y}_1 = \boldsymbol{B}_1$$
$$\boldsymbol{Y}_i = \boldsymbol{A}_i \boldsymbol{Y}_{i-1} + \boldsymbol{B}_i, 2 \leq i \leq n,$$

where the matrices $\boldsymbol{A}_i$ and $\boldsymbol{B}_i$ are $m \times m$.

a. Develop an $O(\log n \log m)$ time algorithm to compute all the $\boldsymbol{Y}_i$'s. What is the total number of operations used? State the PRAM model needed to achieve your complexity bounds.

b. Suppose that $\boldsymbol{B}_i = \boldsymbol{I}$ and $\boldsymbol{A}_i = \boldsymbol{A}$, for $1 \leq i \leq n$. What are the corresponding complexity bounds? What if we want to compute $\boldsymbol{Y}_n$ only?

8.3. Consider the linear system $Ax = d$, where A is an $n \times n$ tridiagonal matrix given by

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & \cdots & & 0 \\ a_2 & b_2 & c_2 & 0 & \cdots & & 0 \\ 0 & a_3 & b_3 & c_3 & \cdots & & 0 \\ \vdots & & \ddots & \ddots & \ddots & & \vdots \\ & & & \ddots & \ddots & \ddots & \\ & & & & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & \cdots & & & 0 & a_n & b_n \end{bmatrix},$$

such that $b_i \neq 0, 1 \leq i \leq n$.

a. Express the solution of this system as a linear recurrence of order 2.

b. Develop an $O(\log n)$ time algorithm to determine x, using $O(n)$ operations.

8.4. Let A be an $n \times n$ tridiagonal matrix as defined in Exercise 8.3. Let d_i be the determinant of the principal submatrix defined by the first i rows and i columns.

a. Show that $d_i = b_i d_{i-1} - a_i c_{i-1} d_{i-2}$, for $2 \leq i \leq n$, where $d_0 = 1$, and $d_1 = b_1$.

b. Let $e_i = c_i d_{i-1}$, where $1 \leq i \leq n$. Show that

$$\begin{bmatrix} d_i \\ e_i \end{bmatrix} = \begin{bmatrix} b_i & -a_i \\ c_i & 0 \end{bmatrix} \begin{bmatrix} d_{i-1} \\ e_{i-1} \end{bmatrix}.$$

c. Develop an $O(\log n)$ time algorithm to compute all the d_i's. Your algorithm should use $O(n)$ operations.

8.5. Let A be an $n \times n$ unit lower triangular matrix defined by

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ a_{21} & 1 & 0 & \cdots & 0 \\ a_{31} & a_{32} & 1 & & 0 \\ \vdots & & & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & & 1 \end{bmatrix}.$$

Define the matrix $E_i, 1 \leq i \leq n$, as follows:

$$\begin{bmatrix} 1 & 0 & \cdots & & & & 0 \\ 0 & 1 & \cdots & & & & \\ \vdots & & \ddots & & & & \\ 0 & 0 & \cdots & 1 & & & \\ 0 & 0 & \cdots & a_{i+1,i} & 1 & & \\ \vdots & & & \vdots & & \ddots & \\ 0 & \cdots & 0 & a_{n,i} & 0 & \cdots & 1 \end{bmatrix}.$$

a. Show that $A = E_1 E_2 \cdots E_n$.
b. Specify the inverse of E_i.
c. Deduce an $O(\log^2 n)$ time algorithm to compute A^{-1}. What is the total number of operations used?

8.6. Let A be an $n \times n$ unit lower triangular matrix, and let $A = I - L$, where L is strictly lower triangular.
a. Show that $L^n = 0$.
b. Deduce that the solution of the linear system $Ax = b$ is given by $x = (I + L + L^2 + \cdots + L^{n-1})b$. What are the complexity bounds of the corresponding algorithm?

8.7. Rewrite the FFT algorithm (Algorithm 8.2) by removing the recursion. Carefully describe the algorithm to be performed by processor P_i, where $0 \le i \le p - 1$ and $p < n$ is a power of 2. Show how the different powers of ω are computed.

8.8. Let W_n be the $n \times n$ matrix defining the DFT over the complex field.
a. Show that the inverse of W_n is given by $W_n^{-1}(j, k) = \frac{1}{n}\omega^{-jk}$, for $0 \le j, k \le n - 1$.
b. Develop an FFT-type algorithm to compute the inverse DFT defined by $y = W_n^{-1}x$. What are the complexity bounds of your algorithm?

8.9. Let $(R, +, .)$ be a commutative ring such that ω is a primitive nth root of unity. Show that the FFT algorithm works correctly over R.

8.10. The purpose of this exercise is to derive another FFT algorithm to compute the DFT $y = W_n x$, assuming that n is a power of 2.
a. Show that

$$y_k = \sum_{j=0}^{\frac{n}{2}-1} \omega^{2jk} x_{2j} + \omega^k \sum_{j=0}^{\frac{n}{2}-1} \omega^{2jk} x_{2j+1}, \quad 1 \le k \le n.$$

b. Deduce an FFT algorithm to compute y. Your algorithm should run in $O(\log n)$ time, using a total of $O(n \log n)$ operations.
c. Draw the flowgraph of your algorithm for $n = 8$, and compare it with Algorithm 8.2.
d. Interpret your algorithm in terms of evaluating a polynomial at the n distinct roots of unity.

8.11. **a.** Show that the product of two $n \times n$ lower triangular Toeplitz matrices is a lower triangular Toeplitz matrix.
b. Deduce an $O(\log n)$ time algorithm to multiply lower triangular Toeplitz matrices. Your algorithm must use $O(n \log n)$ operations.
c. Is the product of two arbitrary Toeplitz matrices Toeplitz? How quickly can you multiply such matrices, and what is the corresponding total number of operations?

8.12. Let $p_1(x), \ldots, p_s(x)$ be polynomials over a ring that can support the FFT algorithm, with each polynomial of degree at most n. Develop an $O(\log (sn))$ time algorithm to compute the product of the p_i's. What is the total number of operations used?

8.13. **a.** Let $p(x_1, x_2)$ and $q(x_1, x_2)$ be two polynomials over the complex field such that the degree of each polynomial in x_1 or in x_2 is $< n$. Use the FFT algorithm to compute the product $p(x_1, x_2)q(x_1, x_2)$ in $O(\log n)$ time, using a total of $O(n^2 \log n)$ operations. Assume polynomials p and q are given in the following form: $p(x_1, x_2) = \sum_{j=0}^{n-1} p_j(x_2)x_1^j$ and $q(x_1, x_2) = \sum_{j=0}^{n-1} q_j(x_2)x_1^j$.

b. *Generalize your algorithm to the case when the number of variables is a parameter m.

8.14. Let $\boldsymbol{a} = [a_0, \ldots, a_{n-1}]$ and $\boldsymbol{b} = [b_0, \ldots, b_{n-1}]$ be two vectors over a field that can support the FFT algorithm. The **positive folded convolution** is a vector $\boldsymbol{c} = [c_0, \ldots, c_{n-1}]$ such that $c_j = \sum_{k=0}^{j} a_k b_{j-k} + \sum_{k=j+1}^{n-1} a_k b_{n+j-k}$, for $0 \le j \le n-1$. Show how to compute the positive folded convolution in $O(\log n)$ time, using a total of $O(n \log n)$ operations.

8.15. Let $p_1, p_2, \ldots, p_n$ be n elements from a field $\mathcal{F}$ that can support the FFT algorithm. The **elementary symmetric functions** of the p_i's are defined as follows:

$$\sigma_1 = \sum_{i=1}^{n} p_i$$

$$\sigma_2 = \sum_{i<j} p_i p_j$$

$$\sigma_3 = \sum_{i<j<k} p_i p_j p_k$$

$$\vdots$$

$$\sigma_n = p_1 p_2 \cdots p_n.$$

Develop an $O(\log^2 n)$ time algorithm to compute the elementary symmetric functions. What is the total number of operations used? *Hint*: Relate the elementary symmetric functions to the coefficients of the polynomial $(x - p_1)(x - p_2)\cdots(x - p_n)$.

8.16. Let $\boldsymbol{C}$ be an $n \times n$ circulant matrix defined by the vector $\boldsymbol{a} = [a_0, \ldots, a_{n-1}]$, and let $p(x) = \sum_{i=0}^{n-1} a_i x^i$.

a. Show that $\boldsymbol{C}\boldsymbol{v} = p(\omega)\,\boldsymbol{v}$, where $\boldsymbol{v}$ is the vector $\boldsymbol{v} = [1, \omega, \omega^2, \ldots, \omega^{n-1}]^T$ and ω is a primitive nth root of unity.

b. Extend the result of part (a) to show that the $p(\omega^j)$'s, for $0 \le j \le n-1$, are the eigenvalues of $\boldsymbol{C}$. Determine the corresponding eigenvectors.

c. Using the facts shown in part (b), establish the correctness of Theorem 8.10.

8.17. Let $H_t = (h_{ij})$ be the matrix defined by

$$h_{ij} = \begin{cases} 1 \text{ if } j = i + 1; \\ t \text{ if } i = n, j = 1; \\ 0 \text{ otherwise.} \end{cases}$$

Let $A_t = \sum_{i=0}^{n-1} a_i H_t^{i-1}$. Then, A_t is always a Toeplitz matrix such A_0 is upper triangular and A_1 is circulant.

a. Show that $H_t = \delta \Delta H_1 \Delta^{-1}$, where Δ is the diagonal matrix $\Delta = diag\{1, \delta, \delta^2, \ldots, \delta^{n-1}\}$, and $\delta^n = t$.

b. Let B be a circulant matrix defined by the vector b. Show that $A_t = \Delta B \Delta^{-1}$ and $b = \Delta a$.

c. Let $t \neq 0$ and $\det(A_t) \neq 0$. Develop an algorithm to solve the system $A_t x = b$ in $O(\log n)$ time, using a total of $O(n \log n)$ operations.

8.18. Let $f(x)$ be a function of a variable x. **Newton's iterative method** to find a zero r of $f(x)$ consists of the iterative scheme $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}, i \geq 1$, where x_0 is an initial approximation to r.

Let $p(x) = \sum_{i=0}^{n} a_i x^i$ be a polynomial of degree n such that $a_0 \neq 0$. We wish to compute the inverse polynomial $q(x) = \frac{1}{p(x)} \bmod x^m$, for some integer m.

a. Apply Newton's method to the equation $f(y) = \frac{1}{y} - p(x)$ to derive the iterative scheme $y_{i+1} = y_i(2 - y_i p(x))$.

b. Let $y_0 = \frac{1}{a_0}$. Show that $\frac{1}{p(x)} - y_{i+1} = \frac{1}{p(x)}\left(\frac{1}{p(x)} - y_i\right)^2$, and deduce that $y_i(x) = q(x) \bmod x^{2^i}$. Develop an $O(\log n \log m)$ time algorithm to compute $q(x)$. What is the total number of operations used?

c. Compare your algorithm with the polynomial division algorithm presented in the text.

8.19. Let $p(x)$ be a polynomial of degree n, and let $q_i(x)$, for $1 \leq i \leq t$, be a set of polynomials such that $n < \sum_{i=1}^{t} \deg(q_i(x)) = d$. Develop a parallel algorithm to compute the polynomials $p_i(x) = p(x) \bmod q_i(x)$, for $1 \leq i \leq t$. Use the same strategy as in Algorithm 8.3 (polynomial evaluation). What are the complexity bounds of your algorithm?

8.20. Let ω be a primitive nth root of unity, where $n = 2^k$. Given an index j whose binary representation is $j = j_{k-1} \ldots j_1 j_0$, where $0 \leq j \leq n - 1$, let $f(j)$ denote the integer whose binary representation is given by $f(j) = j_0 j_1 \ldots j_{k-1}$.

a. Show that $\prod_{j=1}^{l+2^m-1}(x - \omega^{f(j)}) = x^{2^m} - \omega^{f(l/2^m)}$.

b. *Deduce an FFT algorithm based on the strategy of Algorithm 8.3 (polynomial evaluation) applied to the n roots of unity. Your algorithm should run in $O(\log n)$ time, using a total of $O(n \log n)$ operations.

8.21. Let A be an $n \times n$ matrix, and let v be an n-dimensional vector over a field $\mathcal{F}$. Suppose that the Krylov matrix associated with A and v is known to be nonsingular. Show that computing the characteristic polynomial of A can be reduced to solving an $n \times n$ linear system of equations.

8.22. Our algorithm to compute the characteristic polynomial of a matrix (Section 8.8.1) was stated under the assumption that the underlying field has characteristic 0. Why was this assumption used? Can you provide an example of a matrix over a finite field where our algorithm would fail? Explain your answer.

8.23. Let $\mathcal{F}$ be a field of characteristic 0, and let A be an $n \times n$ matrix over $\mathcal{F}$. Given the characteristic polynomial of A, show how to compute the *rank* of A. Deduce an $O(\log^2 n)$ time algorithm to compute the rank of A.

8.24. Given the characteristic polynomial of A, develop an $O(\log^2 n)$ time algorithm to compute A^{-1}, whenever A is nonsingular. What is the total number of operations used?

8.25. Prove Lemma 8.4.

8.26. Let $\phi(\lambda) = \sum_{i=0}^{n} c_i \lambda^i$ be the characteristic polynomial of an $n \times n$ matrix A over a field $\mathcal{F}$. Let $\lambda_1, \lambda_2, \ldots, \lambda_n$ be the roots of $\phi(\lambda)$ (in an extension of $\mathcal{F}$ if necessary). Show that $\sum_{i=1}^{n} \lambda_i^k = tr(A^k)$, for $1 \leq k \leq n - 1$.

8.27. Let $p(x) = \sum_{i=0}^{n} a_i x^i$ be a polynomial of degree n over a field $\mathcal{F}$ such that $a_n = 1$, and let $\lambda_1, \lambda_2, \ldots, \lambda_n$ be the roots of $p(x)$. Define $s_k = \sum_{i=1}^{n} \lambda_i^k$. Prove **Newton's identities:**

$$s_k + \sum_{i=1}^{k-1} a_{n-i} s_i + k a_{n-k} = 0, \quad 1 \leq k \leq n.$$

Hint: $p'(x) = nx^{n-1} + (n-1)a_{n-1}x^{n-2} + \cdots + a_1$, and, on the other hand, $p'(x) = [\prod_{i=1}^{n}(x - \lambda_i)]' = \sum_{i=1}^{n} \frac{p(x)}{x - \lambda_i}$. Substitute the series $\frac{1}{x}\sum_{k \geq 0}\left(\frac{\lambda_i}{x}\right)^k$ for $\frac{1}{x - \lambda_i}$. Compare the two expressions of $p'(x)$.

8.28. Let $\lambda_1, \lambda_2, \ldots, \lambda_n$ be n elements from a field $\mathcal{F}$. Develop an $O(\log^2 n)$ time algorithm to compute $s_k = \sum_{i=1}^{n} \lambda_i^k$, for $1 \leq k \leq n$. Your algorithm should use $O(n \log^2 n)$ operations.

8.29. Develop an $O(\log^3 n)$ time algorithm to compute the QR factorization of a given $n \times n$ nonsingular matrix A. Your algorithm should use $O(nM(n))$ operations. *Hint*: Relate the QR factors of A to the LU factors of $A^T A$.

8.30. Let A be an $n \times n$ matrix whose characteristic polynomial is given by $\phi(\lambda) = \sum_{i=0}^{n} c_i \lambda^i$, and let $s_k = tr(A^k)$, $1 \leq k \leq n$. Let $g(z) = 1 + \sum_{i=1}^{n} c_{n-i} z^i$.

a. Show that $\frac{g'(z)}{g(z)} = -\sum_{j=1}^{k} s_j z^{j-1} \bmod z^k$. *Hint*: Note that $g(z) = \prod_{j=1}^{n}(1 - z\lambda_j)$, where the λ_j's are the eigenvalues of A.

b. Assume that $g_r(z) = g(z) \bmod z^{r+1}$ and the s_i's are available. Show how to compute $g_{2r}(z)$ in $O(\log n)$ time. *Hint*: Express $g_{2r}(z) = g_r(z)h_r(z)$, where $h_r(z) = \sum_{j=r+1}^{2r} b_j z^j$ is to be determined, and show that $\frac{g_r'(z)}{g_r(z)} + h_r'(z) = -\sum_{j=1}^{2r+1} s_j z^{j-1} \bmod z^{2r+1}$.

c. Deduce an $O(\log^2 n)$ time algorithm to compute the c_i's from the s_i's, using a total of $O(n \log n)$ operations.

8.31. Prove the **Cayley–Hamilton theorem** (Section 8.8.1). *Hint*: Start with the identity $(\lambda I - A)\text{Adj}(\lambda I - A) = \det(\lambda I - A)I$, where the adjoint of a matrix Adj is introduced in Exercise 8.32.

8.32. Let R be an arbitrary commutative ring, and let A be an $n \times n$ matrix whose entries come from R. The goal of this exercise (as well as that of Exercise 8.33) is to develop a fast parallel algorithm to compute the characteristic polynomial of A. The algorithm presented in Section 8.8.1 requires division, and hence will not work correctly over an arbitrary commutative ring.

Recall that the **adjoint** of A, denoted by $\text{Adj}(A)$, is an $n \times n$ matrix whose (i,j) entry is equal to $(-1)^{i+j} \det(A_{ji})$, where A_{ji} is the $(n-1) \times (n-1)$ matrix we obtain from A by deleting the jth row and the ith column.

Partition A as follows:

$$A = \begin{bmatrix} a_{11} & u \\ v^T & M \end{bmatrix},$$

where u and v are vectors. Let $p(\lambda) = \det(\lambda I - A) = \sum_{i=0}^{n} p_{n-i}\lambda^i$, and let $q(\lambda) = \det(\lambda I - M) = \sum_{i=0}^{n-1} q_{n-i-1}\lambda^i$.

a. Show that $p(\lambda) = (\lambda - a_{11})\det(\lambda I - M) + u\text{Adj}(\lambda I - M)v^T$.

b. Show that $\text{Adj}(\lambda I - M) = \sum_{k=2}^{n}(q_0 M^{k-2} + \cdots + q_{k-2} I)\lambda^{n-k}$.

c. Develop an $O(\log^2 n)$ time algorithm to compute $uM^i v^T$, for $0 \le i \le n-2$. What is the total number of operations used?

d. Deduce an $O(\log^2 n)$ time algorithm to compute the characteristic polynomial of A over R. The total number of operations used should be $O(nM(n))$.

8.33. This exercise covers **Chistov's algorithm** to compute the characteristic polynomial *without using division*.

Let A be an $n \times n$ matrix, and let A_i be the $i \times i$ submatrix consisting of the last i rows and columns of A, $1 \le i \le n$.

a. Show that $\det(A) = 1/\prod_{i=1}^{n}(A_i^{-1})_{11}$, where all the A_i's are assumed to be nonsingular.

b. Deduce that $\det(I - \lambda A_i) = 1/\prod_{j=1}^{i}[(I - \lambda A_j)^{-1} \bmod \lambda^{i+1}$.

c. Develop an $O(\log^2 n)$ time algorithm to compute the characteristic polynomial of A based on the identity established in part (b). Your algorithm should use $O(nM(n))$ operations.

8.34. Show how to solve a linear system of equations whose coefficient matrix is an $n \times n$ Toeplitz matrix in $O(\log^2 n)$ time, over an arbitrary field, using a total of $O(n^3 \log^2 n)$ operations. *Hint*: Use Chistov's algorithm, described in Exercise 8.33, and Newton's iteration algorithm (Algorithm 8.5).

8.35. Let A be an $n \times n$ matrix over a field $\mathcal{F}$ of characteristic 0, and let $v = [1, \lambda, \lambda^2, \ldots, \lambda^{n-1}]$, where λ is an indeterminate.

a. Develop an $O(\log^2 n)$ time algorithm to compute the corresponding Krylov matrix. What is the total number of operations used?

b. Show how to deduce $s_k = tr(A^k)$ for all $1 \leq k \leq n - 1$ from the Krylov matrix computed in part (a).

c. Deduce an $O(\log^2 n)$ time algorithm to compute the characteristic polynomial of A. Compare your algorithm with the algorithm presented in Section 8.8.1.

d. What are the complexity bounds of your algorithm if the matrix A is Toeplitz?

8.36. Let V be an $n \times n$ Vandermonde matrix $V = (v_{ij})$ corresponding to the vector $p = [p_0, \ldots, p_{n-1}]$.

a. Show that $\det(V) = \prod_{0 \leq j < i \leq n} (p_i - p_j)$.

b. Deduce an $O(\log n)$ time parallel algorithm to compute the determinant. What is the total number of operations used?

8.37. Let A be the $n \times n$ symmetric tridiagonal matrix

$$\begin{bmatrix} b_1 & a_2 & 0 & 0 & \ldots & 0 & 0 & 0 \\ a_2 & b_2 & a_3 & 0 & \ldots & 0 & 0 & 0 \\ 0 & a_3 & b_3 & a_4 & \ldots & 0 & 0 & 0 \\ \vdots & & \ddots & \ddots & \ddots & & & \\ & & & \ddots & \ddots & \ddots & & \\ & & & & & & & \\ 0 & & & & & a_{n-1} & b_{n-1} & a_n \\ 0 & \ldots & & & & 0 & a_n & b_n \end{bmatrix}.$$

Let $p_k(\lambda) = \det(\lambda I - A_k)$, where A_k is the $k \times k$ submatrix consisting of the first k rows and the first k columns.

a. Show that $p_k(\lambda) = (\lambda - b_k)p_{k-1}(\lambda) - a_k^2 p_{k-2}(\lambda)$, where $p_0(\lambda) = 1$ and $p_1(\lambda) = \lambda - b_1$.

b. Develop an $O(\log^2 n)$ time algorithm to compute all the $p_i(\lambda)$'s, using $O(n \log n)$ operations.

8.38. Given two polynomials $u(x) = \sum_{i=0}^{n} a_i x^i$ and $v(x) = \sum_{i=0}^{m} b_i x^i$, let $r(x)$ be the monic gcd of $u(x)$ and $v(x)$. Let $\deg(r(x)) = k$, and let $p(x)$ and $q(x)$ be the unique polynomials satisfying $r(x) = p(x)u(x) + q(x)v(x)$.

a. Show that we can assume $\deg(p(x)) \leq m - k - 1$ and $\deg(q(x)) \leq n - k - 1$. *Hint*: Consider the remainder sequence associated with the polynomials $u(x)$ and $v(x)$.

b. Show that, for any $k' > k$, the corresponding resultant system associated with the gcd computation has a solution (Lemma 8.6).

8.39. Let $u(x)$ and $v(x)$ be two polynomials of degrees n and m, respectively, over an algebraically closed field $\mathcal{F}$. Develop a fast parallel algorithm to determine whether or not $u(x)$ and $v(x)$ have a common zero. State the total number of operations used.

8.40. Let $u(x)$ be a monic polynomial of degree n over a field $\mathcal{F}$. Then, $u(x)$ can be expressed uniquely in the form $u(x) = p_1(x)^{e_1} p_2(x)^{e_2} \cdots p_t(x)^{e_t}$, where the $p_i(x)$'s are distinct, monic, irreducible polynomials over $\mathcal{F}$. Develop an $O(\log^2 n)$ time algorithm to test whether there exists an index e_j such that $e_j > 1$. Your algorithm should use $O(n^2 \log^2 n)$ operations.

Bibliographic Notes

Parallel algorithms for linear recurrences have been described by many researchers. An early reference using a strategy similar to the one presented in the text is [27]. Further early developments were reported in [24, 34, 35]. Parallel algorithms for polynomial evaluation were discovered independently of those for linear recurrences [8, 38]. The divide-and-conquer algorithm for solving triangular linear systems is from [8], whereas the two other algorithms mentioned in Exercises 8.5 and 8.6 are from [44] and [25], respectively. Efficient algorithms for banded triangular systems were discussed in [12, 25]. The fast Fourier transform algorithm was made popular by Cooley and Tukey [15]; the basic algorithm was known earlier (see [16]). Its relevance to convolution was pointed out in [48]. References [6] and [36] cover many FFT algorithms for computing DFT and convolution. The algorithms for polynomial evaluation and interpolation using the FFT algorithm were developed independently by several researchers (see, for example [7, 18]). The equivalence between polynomial division and the inversion of triangular Toeplitz matrices is explored in [5], together with the correspondence between known algorithms for both problems. The first fast parallel algorithms for computing the inverse of a matrix and its characteristic polynomial were given by Csanky [17]. The basic algorithms had been developed much earlier by Leverrier. More efficient implementations of these algorithms are reported in [20, 43]. Berkowitz's algorithm [2] for computing the characteristic polynomial without using division, outlined in Exercise 8.32, is based on an algorithm of Samuelson [45]. The other algorithm requiring no division (Exercise 8.33) was developed by Chistov [13]. Our LU decomposition algorithm is taken from Pan [39]. Iterative algorithms in numerical computing are used heavily and have a long history. The matrix-inversion algorithm described in the text is a classical one (see, for example [30]). The efficiency of its parallel implementation and other choices for the initial

approximation were given by Pan and Reif [42]. There is a substantial literature on structured matrices and Toeplitz matrices. Our description follows [41]. For the proof of Theorem 8.21 and related results, see [19, 22, 50]. The theory of resultants as it relates to remainder sequences and gcd computations was reported in [11, 14]. For some of the best known sequential algorithms, see [10]. For recent parallel algorithms, see [29, 32]. Concerning remainder sequences and gcd computations, it has been shown that these can be reduced to simple operations on Hankel and Toeplitz matrices [4].

Basic facts about matrices as used in this chapter and many other properties and algorithms can be found in [21, 23, 47]. The facts used from abstract algebra can be found in [26, 31]. For additional related algebraic algorithms, see [9, 32, 33, 51, 52].

Exercise 8.4 is taken from [49]. Exercise 8.17 is taken from [3]. The algorithm referred to in Exercise 8.20 was initially described in [18]; see also [1]. Computing the rank of a matrix (see Exercise 8.23) is an important problem, which is discussed in [28, 37]. Exercises 8.29 and 8.30 are taken from [39] and [40, 46], respectively.

References

1. Aho, A. V., J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.
2. Berkowitz, S. On computing the determinant in small parallel time using a small number of processors. *Information Processing Letters*, 18(3):147–150, 1984.
3. Bini, D. Parallel solution of certain Toeplitz linear systems. *SIAM J. Computing*, 13(2):268–276, 1984.
4. Bini, D., and L. Gemignani. On the Euclidean scheme for polynomials. In *Proceedings Second Annual Symposium Parallel Algorithms and Architectures*, Island of Crete, Greece, 1990, pp. 254–258.
5. Bini, D., and V. Pan. Polynomial division and its computational complexity. *Journal of Complexity*, 2(3):179–203, 1986.
6. Blahut, R. E. *Fast Algorithms for Digital Signal Processing.* Addison-Wesley, Reading, MA, 1985.
7. Borodin, A., and R. Moenck. Fast modular transforms. *Journal of Computer and System Sciences*, 8:366–386, 1974.
8. Borodin, A., and I. Munro. *The Computational Complexity of Algebraic and Numeric Problems.* American Elsevier, New York, 1975.
9. Borodin, A., J. von zur Gathen, and J. Hopcroft. Fast parallel matrix and gcd computation. *Information and Control*, 52(3):241–256, 1982.
10. Brent, R. P., F. Gustavson, and S. Y. Yun. Fast solution of Toeplitz systems of equations and computation of padé approximations. *Journal of Algorithms*, 1(3):259–295, 1980.
11. Brown, W., and J. Traub. On Euclid's algorithm and the theory of subresultants. *JACM*, 18(4):505–514, 1971.
12. Chen, S. C., D. J. Kuck, and A. H. Sameh. Practical parallel band triangular systems solvers. *ACM Transactions on Mathematical Software*, 4(3):270–277, 1978.

13. Chistov, A. Fast parallel calculation of the rank of matrices over a field of arbitrary characteristic. In *Proceedings of FCT, Lecture Notes Computer Science*, Springer Verlag, New York, 1985, pp. 63-69.

14. Collins, G. Subresultants and reduced polynomial remainder sequences. *JACM*, 14(1):128-142, 1967.

15. Cooley, J., and J. Tukey. An algorithm for machine calculation of complex Fourier series. *Mathematics of Computation*, 19(89-92):297-301, 1965.

16. Cooley, J. W., P. A. Lewis, and P. D. Welch. History of the fast Fourier transform. *Proceedings IEEE*, 55(10):1675-1677, 1967.

17. Csanky, L. Fast parallel matrix inversion algorithms. *SIAM J. Computing*, 5(4):618-623, 1976.

18. Fiduccia, C. M. Polynomial evaluation via the division algorithm: The fast Fourier transform revisited. In *Proceedings Fourth Annual ACM Symposium on Theory of Computing*, Denver, CO, 1972, pp. 88-93.

19. Friedlander, B., M. Morf, T. Kailath, and L. Ljung. New inversion formulas for matrices classified in terms of their distance from Toeplitz matrices. *Linear Algebra and Its Applications*, 27(1):31-60, 1979.

20. Galil, Z., and V. Pan. Parallel evaluation of the determinant and of the inverse of a matrix. *Information Processing Letters*, 30:41-45, 1989.

21. Gantmacher, F. R. *The Theory of Matrices.* Chelsea, New York, 1959.

22. Gohberg, I. C., and A. A. Semencul. On the inversion of finite Toeplitz matrices and their continuous analogs. *Mat. Issled.*, 2(24):201-233, 1972.

23. Golub, G., and C. van Loan. *Matrix Computations.* Johns Hopkins University Press, Baltimore, MD, 1989.

24. Heller, D. Some aspects of the cyclic reduction algorithm for block tridiagonal linear systems. *SIAM J. on Numerical Analysis*, 13(4):484-496, 1976.

25. Heller, D. A survey of parallel algorithms in numerical linear algebra. *SIAM Review*, 20(4):740-777, 1978.

26. Herstein, I. N. *Topics in Algebra.* Xerox College Publishing, Lexington, MA, 1964.

27. Hockney, R. W. A fast direct solution of Poisson's equation using Fourier analysis. *JACM*, 12(1):95-113, 1965.

28. Ibarra, O., S. Moran, and L. E. Rosier. A note on the parallel complexity of computing the rank of order n matrices. *Information Processing Letters*, 11(4-5):162, 1980.

29. Ierardi, D., and D. Kozen. Parallel resultant computation. In J. Reif, editor, *Synthesis of Parallel Algorithms.* Morgan Kaufman, San Mateo, CA, 1991.

30. Isacson, E., and H. B. Keller. *Analysis of Numerical Methods.* Wiley, New York, 1966.

31. Jacobson, N. *Basic Algebra I.* W. H. Freeman & Co., San Francisco, 1974.

32. Kaltofen, E. Parallel algebraic algorithm design. Technical report, Computer Science Department, Rensselaer Polytechnic Institute, Troy, NY, 1989.

33. Knuth, D. *The Art of Computer Programming: Seminumerical Algorithms*, second edition. Addison-Wesley, Reading, MA, 1981.

34. Kogge, P. M. Parallel solution of recurrence problems. *IBM Journal of Research and Development*, 18(2):138-148, 1974.

35. Kogge, P. M., and H. S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, 22(8):786–792, 1973.

36. McClellan, J. H., and C. M. Rader. *Number Theory in Digital Signal Processing.* Prentice-Hall, Englewood Cliffs, NJ, 1979.

37. Mulmuley, K. Fast parallel algorithm to compute the rank of a matrix. *Combinatorica*, 7(1):101–104, 1987.

38. Munro, I., and M. Paterson. Optimal algorithms for parallel polynomial evaluation. *Journal of Computer and System Sciences*, 7(2):189–198, 1973.

39. Pan, V. Complexity of parallel matrix computations. *Theoretical Computer Science*, 54(1):65–85, 1987.

40. Pan, V. Parallel least-squares solution of general and Toeplitz-like linear systems. In *Proceedings Second Annual Symposium Parallel Algorithms and Architectures*, Island of Crete, Greece, 1990, pp. 244–253.

41. Pan, V. Parameterization of Newton's iteration for computations with structured matrices and applications. Technical Report CUCS-032-90, Department of Computer Science, Columbia University, New York, 1990.

42. Pan, V., and J. Reif. Efficient parallel solution of linear systems. In *Proceedings Seventeenth Annual Symposium on Theory of Computing*, pages 143–152, Providence, RI, 1985.

43. Preparata, F. P., and D. V. Sarwate. An improved parallel processor bound in fast matrix inversion. *Information Processing Letters*, 7(3):148–149, 1978.

44. Sameh, A., and R. Brent. Solving triangular systems on a parallel computer. *SIAM J. on Numerical Analysis*, 14(6):1101–1113, 1977.

45. Samuelson, P. A method for determining explicitly the coefficients of the characteristic polynomial. *Annals of Mathematical Statistics*, 13(4):424–429, 1942.

46. Schonhage, A. The fundamental theorem of algebra in terms of computational complexity. Manuscript, Department of Mathematics, University of Tubingen, Germany, 1982.

47. Stewart, G. W. *Introduction to Matrix Computations.* Academic Press, Cambridge, MA, 1973.

48. Stockham, T. G. High speed convolution and correlation. In *AFIPS Conference Proceedings*, Boston, MA, 1966, pp. 299–233.

49. Swarztrauber, P. N. A parallel algorithm for solving general tridiagonal equations. *Mathematics of Computation*, 33(145):185–199, 1979.

50. Trench, W. F. An algorithm for inversion of finite Toeplitz matrices. *Journal of SIAM*, 12(3):515–522, 1964.

51. von zur Gathen, J. Parallel arithmetic computations: a survey. *SIAM J. Computing*, 13(4):802–824, 1984.

52. von zur Gathen, J. Parallel arithmetic computation: a survey. In *Mathematical Foundations of Computer Science Proceedings MFCS*, Bratislava, Czechoslovakia, 1986, pp. 93–112.

9

Randomized Algorithms

Thus far, in search of efficient methods to solve our selected problems, we have explored the domain of *deterministic algorithms*. These algorithms are required to generate exact solutions for all possible instances within time bounds that depend on a worst-case analysis. In many situations, this goal seems to impose an unnecessarily rigid framework that detracts from the applicability of the algorithms. There are two alternative approaches that have been studied in the literature to a limited extent. The first approach is based on developing algorithms that work well for typical instances of the problem; an *average-case* analysis is carried out under a certain assumption on the probability distribution of the inputs. The difficulties with this approach lie in determining what constitutes a good distribution of the inputs, and in carrying out the probabilistic analysis of the corresponding algorithm.

The other approach—the one we shall follow in this chapter—is to introduce randomness into the algorithm by using a random-number generator. A classical example is to extract a random sample from the given input, to analyze the sample, and to develop a solution for the given input by using the analysis made on the sample. We may also allow the algorithm to generate incorrect answers with a very small probability. We are interested in such algorithms because, for many problems, they are considerably simpler or faster than are the corresponding deterministic algorithms.

Algorithms making use of random-number generators are called **randomized algorithms.** The analysis of the asymptotic bounds of such algorithms does not make any assumptions about the input. These algorithms may behave differently when run several times on the same input. Their performance will be probabilistically guaranteed for *any* input.

In this chapter, we shall develop randomized parallel algorithms for a variety of problems, including *point location in triangulated planar subdivisions* (Section 9.3), *one- and two-dimensional string matching* (Section 9.4), *verification of polynomial identities* (Section 9.5), *sorting* (Section 9.6), and *maximum matching* (Section 9.7). The analysis of our algorithms will draw on several facts from probability theory, which we summarize in our first section.

9.1 Performance Measures of Randomized Parallel Algorithms

Since a randomized algorithm makes random choices during its execution, we measure its performance in probabilistic terms, such as by its expected running time or by the probability that it will terminate within a certain number of steps. The analysis of such algorithms requires tools borrowed from probability theory and combinatorial mathematics. Section 9.1.1 gives a brief introduction to basic facts from probability theory that will be used in this chapter. In Section 9.1.2, we define the additional assumptions that have to be introduced into our basic parallel model to support randomized algorithms, and we discuss performance measures for randomized parallel algorithms.

9.1.1 PROBABILITY THEORY

We assume that you are familiar with basic facts from probability theory. The purpose of this section is to provide a quick review of the facts you need to follow the material covered in this chapter.

As we introduce the definitions and related facts, we shall apply them to one of two setups: tossing coins and placing balls into bins. These situations are introduced in the following two examples.

EXAMPLE 9.1: (Coin Tossing)

Consider the experiment of tossing a coin n times. The outcome of each coin toss is either a head (H) or a tail (T). Hence, each outcome of the experiment can be represented by a sequence of length n of the symbols H and T. The set

of all possible outcomes forms what is called a *sample space* S. Each of these outcomes constitutes an *elementary event*. Hence, an elementary event can be viewed as a particular sequence of length n of H and T. The *event* that exactly two heads show up corresponds to the subset of the sample space consisting of all the sequences containing exactly two H symbols. Hence, this event is a subset of S consisting of $\binom{n}{2}$ elementary events. □

EXAMPLE 9.2: (Balls and Bins)

Consider the experiment of randomly placing into n bins m balls labeled $1, 2, \ldots, m$. An outcome of this experiment is an arrangement of the balls into the bins, where a bin can hold any number of balls. The *sample space* consists of all such arrangements, and each such arrangement constitutes an *elementary event*. Each outcome can be viewed as assigning to each ball a bin number between 1 and n. Hence, the size of the sample space is n^m. The event that the first two bins are empty corresponds to all the outcomes for which each ball is assigned a bin number between 3 and n. Therefore, this event contains $(n - 2)^m$ elementary events. □

In general, a **sample space** S consists of a set of **elementary events** that represents the outcomes of an experiment. An **event** is simply a subset of S. A sample space S is **discrete** if S is either finite or countably infinite. We shall deal exclusively with discrete sample spaces in this chapter.

A **probability measure** on a discrete sample space S is a real-valued function defined on the subsets of S such that the following axioms are satisfied:

1. For all events A, $0 \leq Pr\{A\} \leq 1$.
2. $Pr\{S\} = 1$.
3. $Pr\{A \cup B\} = Pr\{A\} + Pr\{B\}$, for any two disjoint events (that is, $A \cap B = \emptyset$).

We call the value $Pr\{A\}$ the **probability** of the event A.

EXAMPLE 9.3:

We continue our coin-tossing experiment (Example 9.1). Our sample space S can be represented by the set $\{H, T\}^n$, where each experiment consists of a sequence of n coin tosses. We define the following probability measure on S: For each event A, let $Pr\{A\} = |A|/2^n$. It is easy to verify that all the axioms of a probability measure are satisfied. Intuitively, we are assuming that the coin is *unbiased*, and hence that the outcome of each coin toss is equally likely to be a head or a tail. This assumption implies that each elementary event is equally likely to occur, and hence that each has probability $1/2^n$, since the size of the sample space is 2^n.

Whenever all elementary events are equally likely, the resulting probability measure is referred to as the **uniform probability distribution.** □

Given any two events A and B from a sample space S such that $Pr\{B\} > 0$, the **conditional probability** of A given B is defined to be

$$Pr\{A \mid B\} = \frac{Pr\{A \cap B\}}{Pr\{B\}}.$$

The two events A and B will be called **independent** if $Pr\{A \cap B\} = Pr\{A\}Pr\{B\}$, and hence $Pr\{A \mid B\} = Pr\{A\}$ and $Pr\{B \mid A\} = Pr\{B\}$.

Another important property that we will often use is to bound the probability of the union of a collection of events $\{A_i\}$ by their individual probabilities. By the third axiom of a probability measure, we know that $Pr\{\cup_i A_i\} = \sum_i Pr\{A_i\}$ when the events $\{A_i\}$ are pairwise disjoint. In general, however, we have the following inequality, sometimes referred to as **Boole's inequality:**

$$\Pr\{\bigcup_i A_i\} \leq \sum_i Pr\{A_i\}. \tag{9.1}$$

EXAMPLE 9.4:

We return to the coin-tossing experiment (Examples 9.1 and 9.3). Suppose we toss an unbiased coin three times. The sample space can be represented by the set $S = \{HHH, HHT, HTH, HTT, THH, THT, TTH, TTT\}$. Consider the following two events A and B; event A consists of all tosses resulting in at least two tails, and hence $A = \{HTT, THT, TTH, TTT\}$; event B consists of all tosses resulting in a tail during the first toss, and hence $B = \{THH, THT, TTH, TTT\}$. What is the probability of A given B? That is, what is the probability of getting at least two tails, given that the first toss resulted in a tail? Since $A \cap B = \{THT, TTH, TTT\}$, we obtain that $Pr\{A \mid B\} = 3/4$. Notice that $Pr\{A\} = Pr\{B\} = 1/2$.

On the other hand, since $A \cup B = \{HTT, THH, THT, TTH, TTT\}$, we have $Pr\{A \cup B\} = 5/8$, whereas $Pr\{A\} + Pr\{B\} = 1$. □

EXAMPLE 9.5:

We return to the experiment of randomly placing m balls into n bins (Example 9.2), where m is much larger than n—say, $m \geq 2n \ln n$. As we saw in Example 9.2, there are n^m possible arrangements, each of which will be assumed to be equally likely. We are interested in determining a bound on the probability that two bins (or more) are empty.

Given two distinct integers i and j such that $1 \leq i, j \leq n$, let E_{ij} be the event that bins i and j remain empty. Clearly, $Pr\{E_{ij}\} = (n-2)^m/n^m = \left(1 - \frac{2}{n}\right)^m$. Using Boole's inequality (Eq. 9.1), we obtain

$$Pr\{\bigcup_{i \neq j} E_{ij}\} \leq \binom{n}{2}\left(1 - \frac{2}{n}\right)^m.$$

Given that $1 + x \leq e^x$, this bound can be approximated by

$$Pr\{\bigcup_{i \neq j} E_{ij}\} \leq \binom{n}{2} e^{-\frac{2m}{n}} \leq n^{-2}. \qquad \square$$

Random Variables. A **random variable** X is a function from a sample space S into the field of real numbers. Most of the random variables occurring in randomized algorithms are integer-valued (for example, the number of steps needed to execute a randomized algorithm).

Let the event $X = x$ be the set of all elementary events for which X takes on the value x. Then, the **probability distribution** of X is the function $Pr\{X = x\}$. The **distribution function** of X is defined to be the function $Pr\{X \leq x\}$.

EXAMPLE 9.6:

The number X of heads occurring during the experiment of tossing a coin n times is a random variable. Let us determine its probability distribution. The number of outcomes with exactly x heads is

$$\binom{n}{x},$$

where x is an integer between 0 and n. Assuming that the coin is unbiased, the probability of x heads is

$$\binom{n}{x}\frac{1}{2^n},$$

which defines the probability distribution of X. The distribution function of X is given by

$$Pr\{X \leq x\} = \sum_{j=0}^{x} \binom{n}{j}\frac{1}{2^n}. \qquad \square$$

The **expected** or **mean value** of a discrete random variable X is

$$E[X] = \sum_{x} xPr\{X = x\},$$

where the sum is over all possible values that can be assumed by X. In all the cases encountered in this chapter, X is integer-valued and the preceding sum is finite. We shall usually denote $E[X]$ by the symbol μ_X (and shall drop the X when there is no possibility of confusion). For random variables $X_1, X_2, \ldots, X_k$, we have the following important linearity property of expectation:

$$E\left[\sum_i X_i\right] = \sum_i E[X_i].$$

The **variance** σ_X^2 of the random variable X is defined to be $E[(X - \mu_X)^2]$, which can be shown to be equal to $E[X^2] - \mu_X^2$. The **standard deviation** σ_X is the positive square root of σ_X^2.

The following is an important fact concerning the deviation of a random variable from its expectation.

Theorem 9.1 (Chebyshev Bound): *Given a random variable X with expectation μ and standard deviation σ,*

$$Pr\{|X - \mu| > t\sigma\} \leq \frac{1}{t^2}.$$ □

In randomized algorithms, the Chebyshev bound is sometimes used to estimate the probability that the number of steps that the algorithm takes is significantly more than the expected value.

We next discuss a specific distribution that arises frequently in analyzing randomized algorithms.

The Binomial Distribution. A **Bernoulli trial** is an experiment with only two possible outcomes, one of which is called a **success,** and the other of which is called a **failure.** We shall denote the probability of a success by p, and the probability of a failure by q (and hence $p + q = 1$).

The coin-tossing experiment is a Bernoulli trial where a head could be considered a success and a tail considered a failure. In a randomized iterative algorithm where, after each iteration, the input is supposed to shrink by a factor of 1/2, each iteration could be viewed as a Bernoulli trial where the outcome is successful if the size of the input decreases by a factor of 1/2; otherwise the outcome would be considered a failure.

Let X denote the number of successes during n *independent* Bernoulli trials that have a success probability of p and a failure probability of $q = 1 - p$. The random variable X can take on the values $0, 1, \ldots, n$. Its probability distribution is given by

$$Pr\{X = k\} = \binom{n}{k} p^k q^{n-k},$$

which is called the **binomial distribution,** denoted by $b(k;n, p)$.

EXAMPLE 9.7:

We return to the random placement of m balls into n bins. We seek the probability that a specified bin contains exactly k balls, where $0 \leq k \leq m$. There are

$$\binom{m}{k}$$

ways of choosing the k balls for the specified bin. The remaining $m - k$ balls can be placed in any of the bins other than the specified bin. Hence, the desired probability is given by

$$\binom{m}{k} \frac{(n-1)^{m-k}}{n^m} = \binom{m}{k} \frac{1}{n^k} \left(1 - \frac{1}{n}\right)^{m-k},$$

which is a binomial distribution $b\left(k; m, \frac{1}{n}\right)$. □

We now compute the expected value of the binomial variable X with the probability distribution $b(k; n, p)$. Rather than using the definition of $E[X]$ directly, we develop a simpler method that provides a useful general tool.

Let X_i be a random variable such that $X_i = 1$ if the ith Bernoulli trial results in a success; otherwise, $X_i = 0$. We start by computing the expected value of X_i. Using the definition, we get $E[X_i] = 0 \cdot Pr\{X_i = 0\} + 1 \cdot Pr\{X_i = 1\} = p$. Now the number X of successes among n independent Bernoulli trials is exactly $\sum_{i=1}^{n} X_i$. Hence, $E[X] = E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i] = np$.

Frequently, we are interested in estimating the probability that a binomial variable X is larger (or smaller) than a certain value m. The probability that X is larger than m is given by

$$Pr\{X \geq m\} = \sum_{j=m}^{n} \binom{n}{j} p^j q^{n-j}.$$

The equation for the probability of X smaller than m is similar. The expression shown is referred to as the **tail** of the binomial distribution. The following estimates, called **Chernoff bounds,** will be used extensively:

$$Pr\{X \leq (1 - \epsilon)pn\} \leq e^{-\epsilon^2 np/2} \tag{9.2}$$

$$Pr\{X \geq (1 + \epsilon)pn\} \leq e^{-\epsilon^2 np/3}, \tag{9.3}$$

where ϵ is any positive constant between 0 and 1.

We now digress for a moment to state several important combinatorial facts.

Combinatorial Facts. We state without proof several combinatorial facts that will be used in the remainder of this chapter. Most of these facts can be found in elementary books on combinatorics.

Binomial Theorem: Given any two numbers x and y and a positive integer n, then

$$(x + y)^n = \sum_{i=0}^{n} \binom{n}{i} x^i y^{n-i}. \tag{9.4}$$

Stirling's Formula: Given any positive integer n, then

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n. \tag{9.5}$$

Bounds on Binomial Coefficients: Given any two positive integers n and k such that $k \leq n$, then

$$\left(\frac{n}{k}\right)^k \leq \binom{n}{k} \leq \left(\frac{ne}{k}\right)^k. \tag{9.6}$$

Poisson Approximation: For any α, we have

$$\left(1 - \frac{\alpha}{n}\right)^n \approx e^{-\alpha}. \tag{9.7}$$

We next introduce the *randomized PRAM* model and related performance measures.

9.1.2 RANDOMIZATION IN PARALLEL ALGORITHMS

To support randomized parallel algorithms, we must incorporate a few additional features into our PRAM model. The main feature is to allow each processor the ability to generate a random number in a certain range within a single step. A random number in the integer range $[1, 2, \ldots, M]$ is equally likely to take on any integer value between 1 and M. We restrict the length of the random number generated in a single step to be $O(\log n)$ bits, where n is the length of the input. Hence, each such number can fit in a single memory location and can be manipulated in $O(1)$ sequential steps. We assume that, if k processors generate random numbers during the same time step, then the resulting k numbers are k **independent** random numbers within the chosen range. A PRAM with these additional features will be called a **randomized** PRAM.

The complexity measures of a deterministic parallel algorithm, given in the time-work framework, are expressed in terms of the time $T(n)$ and of the total number of operations $W(n)$. We can take the **expected** values of $T(n)$ and $W(n)$ on any input of size n as the corresponding complexity measures for randomized parallel algorithms. A major weakness in dealing with expected values is that the actual resources used may be far different from these values for many cases. We would rather establish **high-likelihood bounds,** as defined next.

A randomized parallel algorithm A requires resource bound $f(n)$ with **high probability** if, for any input of size n, the amount of the resource used by algorithm A is at most $\alpha f(n)$ with probability $1 - n^{-c}$, for some positive constants α and c. This definition implies that algorithm A, when run on any input of size n, is *highly likely* to use the amount $O(f(n))$ of that resource.

We distinguish between two classes of randomized algorithms: **Las Vegas** and **Monte Carlo.** A randomized algorithm of the Las Vegas type will always generate the correct answer. Its performance is measured in terms of the expected amount of resources used or the probability that a certain resource bound is exceeded. Las Vegas algorithms are tailored to work well on random instances. A randomized algorithm of the Monte Carlo type is allowed to make errors, but only with a "small" probability. Note that there are more elaborate ways to define Monte Carlo algorithms, but the given definition is sufficient for our purposes.

Both Las Vegas and Monte Carlo algorithms will be illustrated in this chapter.

The next section presents a simple randomization technique for breaking symmetry as applied to the problem of the fractional independent set.

9.2 The Problem of the Fractional Independent Set

Let $G = (V, E)$ be a planar graph. That is, G can be drawn on the plane such that each vertex $v \in V$ is represented by a point $p(v)$, each edge $(u, v) \in E$ is represented by a curve connecting the two points $p(u)$ and $p(v)$, and no two curves that represent edges can intersect except possibly at an endpoint. In this section, we consider the problem of identifying a **large independent set** of G consisting exclusively of **low-degree vertices.** More precisely, our aim is to determine a set $X \subseteq V$ such that (1) the degree of each vertex $v \in X$ is less than or equal to some constant d, (2) the set X is independent, that is, no two vertices of X are connected by an edge, and (3) the size of X satisfies $|X| \geq c|V|$ for some positive constant c. We call X a **fractional independent set** of G.

We shall establish the existence of a fractional independent set in any planar graph by an explicit construction. We need the following well-known fact from graph theory.

Lemma 9.1: *Let $G = (V, E)$ be a planar graph with at least three vertices. Then, $|E| \leq 3|V| - 6$.* □

Lemma 9.1 indicates that planar graphs are quite sparse. Hence, we expect to find many vertices of low degree, since $\sum_{v \in V} \deg(v) = 2|E| \leq 6|V| - 12$. The following theorem gives a constructive proof of the fact that a fractional independent set exists in any planar graph.

Theorem 9.2: *For any given planar graph $G = (V, E)$, we can construct a fractional independent set X in linear sequential time.*

Proof: Let V_d be the set of vertices of G of degree less than or equal to d, where d is any integer such that $d \geq 6$. We start by showing that the size of V_d is at least a constant fraction of $|V|$.

Let V_h be the subset of vertices each of degree larger than d; that is, $V_h = V - V_d$. Then, $\sum_{v \in V_h} \deg(v) \geq (d + 1)|V_h|$, and hence $(d + 1)|V_h| \leq \sum_{v \in V} \deg(v) \leq 6|V| - 12$, which implies that $|V_h| \leq (6|V| - 12)/(d + 1)$. This inequality in turn gives us $|V_d| = |V| - |V_h| \geq (d - 5)|V|/(d + 1)$.

We construct a fractional independent set X as follows. We select an arbitrary vertex $v \in V_d$ to be in X, and remove all of its adjacent vertices that are in V_d. We then pick another vertex of V_d, put it in X, and remove all of its adjacent vertices. We continue this process until all the elements of V_d are exhausted. The number of vertices in X is at least $|V_d|/(d + 1) \geq (d - 5)|V|/(d + 1)^2$, which is a constant fraction of $|V|$, because d is a constant greater than or equal to 6. □

The scheme sketched in the proof of Theorem 9.2 for constructing a fractional independent set picks its vertices in sequence, each selected vertex depending on the vertices selected earlier. Hence, this scheme does not seem to offer much hope for a significant amount of parallelism. In this section, we develop an extremely simple alternate method that uses randomization. The resulting algorithm determines, with high probability, a fractional independent set in $O(1)$ time, using $O(n)$ operations.

For clarity, we start with the simple case where the graph G is a simple directed cycle. We later address the general case of planar graphs.

9.2.1 DIRECTED CYCLES

Let $G = (V, E)$ be a directed cycle whose arcs are represented by an array S of length $|V| = n$ such that $S(i) = j$ if and only if $\langle i, j \rangle$ is an arc in E (see Section 2.7). We wish to compute an independent set X of G such that the size of X is a constant fraction of $|V|$.

The basic technique used is a **randomized symmetry breaking** that consists of (1) assigning a label 1 or 0 with equal probability to each vertex $v \in V$, and (2) relabeling each vertex v whose label and the label of its successor $S(v)$ are both equal to 1.

We state the algorithm more formally next. We shall show that, with high probability, the set of vertices labeled 1 is a fractional independent set of the given directed cycle.

ALGORITHM 9.1
(Randomized Symmetry Breaking)
Input: *A directed cycle $G = (V, E)$ whose arcs are specified by an array S of length n.*
Output: *The set of vertices $X = \{v \in V \mid label(v) = 1\}$, which forms a fractional independent set with high probability.*
begin
  for all $v \in V$ **pardo**
    1. *Assign $label(v) = 1$ or 0 randomly with equal probability.*
    2. **if** ($label(v) = 1$ **and** $label(S(v)) = 1$) **then** set $label(v) := 0$
end

Lemma 9.2: *Let $G = (V, E)$ be a directed cycle with n vertices. Algorithm 9.1 identifies an independent set X such that $Pr\{|X| \leq \alpha n\} \leq e^{-\beta n}$, where α is any constant $0 < \alpha < 1/8$ and $\beta = (1 - 8\alpha)^2/16$. The algorithm runs in $O(1)$ time, using $O(n)$ operations.*

Proof: A vertex $v \in V$ is called *selected* if $label(v) = 1$ when the algorithm terminates. Let X be the set of selected vertices. The set X consists precisely of all the vertices v such that $label(v) = 1$ and $label(S(v)) = 0$ just after the random assignment performed at step 1. Hence, the probability that a vertex $v \in V$ is selected is equal to $Pr\{label(v) = 1 \text{ and } label(S(v)) = 0\} = 1/4$, where the labels referred to are those obtained immediately after the execution of step 1.

Consider the set $V' \subseteq V$ consisting of every other vertex along the cycle, assuming without loss of generality that n is even. Hence, the distance between any two vertices of V' is at least 2. Given any two vertices v and w in V', the event that v is selected is independent of the event that w is selected. Hence, the cardinality of the set X is bounded below by a binomial variable, where the success probability is 1/4 and there are $n/2$ Bernoulli trials. Using the estimate of the binomial tail (Chernoff bounds),

$$Pr\{|X| \leq (1 - \epsilon)\mu\} \leq e^{-\epsilon^2\mu/2},$$

as given by Eq. 9.2, where $\mu = n/8$, we obtain that $Pr\{|X| \leq \alpha n\} \leq e^{-\beta n}$, where $\beta = (1 - 8\alpha)^2/16$ for any $0 < \alpha < 1/8$.

The complexity bounds of the algorithm are obvious. □

PRAM Model: Algorithm 9.1 can be implemented on the EREW PRAM model, since step 1 does not require any concurrent memory access, and step 2 can be executed if we first create a duplicate copy of the labels generated at step 1 and then access this copy whenever the label of the successor node is needed. □

9.2.2 PLANAR GRAPHS

We now return to the more general case when the input graph $G = (V, E)$ is an arbitrary planar graph. We assume that the embedded graph G is represented by its edge lists such that, for each vertex $v \in V$, the edge list of v contains the edges incident on v arranged in the order as they appear counterclockwise around v.

For the remainder of this section, we define a *low-degree vertex to be a vertex of degree less than or equal to 6*. As can be deduced from the proof of Theorem 9.2 (if we set $d = 6$), there are at least $|V|/7$ such vertices.

To determine a fractional independent set of G, we follow the same strategy as in the case of directed cycles. We randomly assign a label of 0 or 1 with equal probability to each low-degree vertex. The set of vertices whose labels are each equal to 1 does not necessarily form an independent set. For each edge $(u, v) \in E$, we relabel vertices u and v whenever u and v were assigned the label 1. The remaining vertices with label 1 constitute an independent set whose size is a constant fraction of $|V|$ with high probability.

We next give the algorithm formally, followed by a correctness proof and an analysis of the complexity bounds.

ALGORITHM 9.2

(Fractional Independent Set)

Input: *A planar graph $G = (V, E)$ represented by its edge lists. The edges on the list of a vertex v are ordered counterclockwise as they appear around the vertex v in a planar embedding of G.*

Output: *A labeled set of low-degree vertices forming a large independent set with high probability.*

begin

1. **for** *each vertex* $v \in V$ **pardo**
   if $(\deg(v) \leq 6)$ **then** *set* $lowdeg(v) := 1$
   else *set* $lowdeg(v) := 0$
2. **for** *each vertex* $v \in V$ **pardo**
   if $(lowdeg(v) = 1)$ **then** *Randomly assign* $label(v) := 0$ *or* 1 *with equal probability.*
   else *set* $label(v) := 0$
3. **for** *each vertex* $v \in V$ **pardo**
   if $(label(v) = 1)$ **then**
   if $(label(u) = 1$ *for some* u *on the list of* $v)$ **then** *set* $label(v) := 0$

end

Theorem 9.3: *The set* $X = \{v \in V \mid label(v) = 1\}$ *generated when Algorithm 9.2 terminates is an independent set of low-degree vertices such that* $Pr\{|X| \leq \alpha n\} \leq e^{-\beta n}$, *for some positive constants* α *and* β *and where* n *is the number of vertices. This algorithm can be implemented in* $O(1)$ *time, using* $O(n)$ *operations.*

Proof: As we can deduce from the proof of Theorem 9.2 (by setting $d = 6$), the number of vertices of degree less than or equal to 6 is at least $n/7$. Each such vertex is assigned a label of 0 or 1 with probability 1/2. Hence the probability of a low-degree vertex being included in the independent set X is at least $(1/2)^7$, since it will be included in X if its label is equal to 1 and the labels of all of its adjacent vertices are equal to 0 just after the execution of step 2.

Let V' be the subset of low-degree vertices such that the distance between any two vertices in V' is at least 3. The size of V' satisfies $|V'| \geq (1/36)(n/7)$, since each vertex included in V' prevents the inclusion of all vertices at distance 2 or less, and the number of such vertices in V' is at most 36. Hence, $|V'| \geq cn$, where c is some positive constant (in this case, $c = 1/252$).

The event that a vertex v from V' is included in X is independent of the event that any other vertex from V' is included in X. Therefore, the cardinality

of the set X is lower bounded by a binomial variable whose probability of success is $(1/2)^7$, and there are cn independent Bernoulli trials. Using Chernoff bounds as stated in Eq. 9.2, we obtain $Pr\{|X| \leq \alpha n\} \leq e^{-\beta n}$, for some constants α and β.

We now analyze the complexity bounds of the algorithm. In step 1, we can count, for each list associated with a vertex v, the number of the edges up to 6. If, during the counting process, the end of the list is encountered, we set $lowdeg(v)$: $= 1$; otherwise, we set $lowdeg(v)$: $= 0$. Given our input representation, this step can be implemented in $O(1)$ time, using a linear number of operations. Step 2 trivially takes $O(1)$ time, using $O(n)$ operations. As for step 3, any vertex v with $label(v) = 1$ must be of degree less than or equal to 6. Hence, its corresponding list is of length at most 6, which implies that each such list can be processed in $O(1)$ sequential time to check for the existence of a vertex u whose label is equal to 1. Therefore, step 3 also can be performed in $O(1)$ time, using a linear number of operations, and the proof of the theorem is complete. □

PRAM Model: Steps 1 and 2 of Algorithm 9.2 require no simultaneous memory access. Since many edges could share the same vertex, step 3 requires concurrent-read capability. However, no concurrent write instructions are used. Therefore, Algorithm 9.2 runs on the CREW PRAM model. □

Based on Algorithm 9.2, we next develop an elegant solution to an important problem in computational geometry.

9.3 Point Location in Triangulated Planar Subdivisions

The randomized algorithm for determining a fractional independent set of a planar graph (Algorithm 9.2) will be used in this section to solve the point-location problem for triangulated planar subdivisions. Before giving a formal definition of our problem, we note that, in general, the technique of **random sampling** has been shown to be effective in solving computational-geometry problems. This technique will be introduced in Section 9.6 for designing a randomized sorting algorithm. Here, we illustrate the use of the randomized symmetry-breaking technique to solve our special computational-geometry problem.

We now introduce the problem addressed in this section. A **planar subdivision** S is a straight-line embedding of a planar graph $G_S = (V_S, E_S)$. Hence, each edge (u, v) is represented by a line segment connecting the points representing u and v, and no two such line segments intersect except

possibly at an endpoint. Because of the close correspondence between a planar subdivision and its associated graph, we shall refer to the points of S as *vertices* and to the segments of S as *edges*.

The **size** of a subdivision S is the number of vertices in S. A **triangulated** planar subdivision is a planar subdivision in which each region, including the exterior region, is a triangle.

EXAMPLE 9.8:

The planar subdivision shown in Fig. 9.1(a) contains regions that are not triangles, whereas all the regions defined by the planar subdivision of Fig. 9.1(b) are triangles. □

Given a triangulated planar subdivision S, the **point-location problem** is to preprocess S and to set up appropriate data structures so that any query point can be located quickly in one of the regions of S. More precisely, we wish to identify the region containing a query point in $O(\log n)$ sequential time.

We shall describe a solution based on building a hierarchy of increasingly coarse subdivisions, starting with the initial subdivision S and ending with a subdivision containing only three vertices. The location of a query point will proceed through the hierarchy consecutively, starting with the

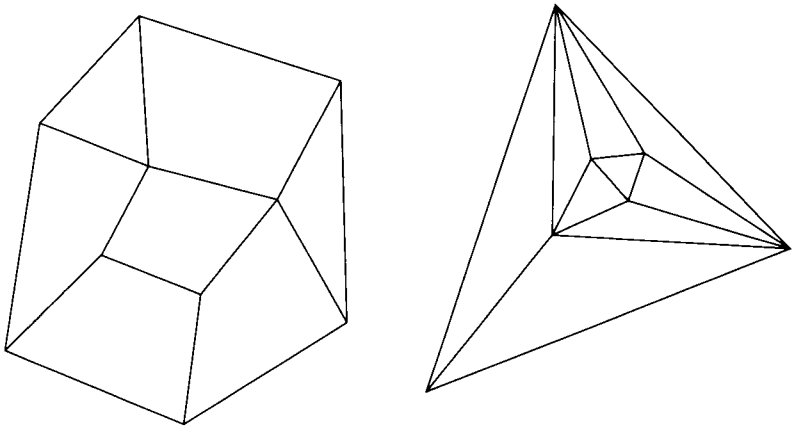


FIGURE 9.1
Planar subdivisions (a) A planar subdivision containing regions that are not triangles, and (b) a triangulated planar subdivision.

coarsest subdivision (which contains three vertices) until the initial subdivision is reached.

In this section, we show how to construct a subdivision hierarchy in $O(\log n)$ time, with high probability such that a query can be handled in $O(\log n)$ sequential time. We start by introducing a formal notion of subdivision hierarchy.

9.3.1 SUBDIVISION HIERARCHY

Let S be a triangulated subdivision with n vertices. A **subdivision hierarchy** is a sequence $S_1, S_2, \ldots, S_{h(n)}$ of triangulated subdivisions satisfying the following conditions:

1. $S_1 = S$.
2. $|S_{h(n)}| = 3$.
3. Each region of S_{i+1} intersects at most d regions of S_i, for some constant d.

We call $h(n)$ the **height** of the subdivision hierarchy. A subdivision hierarchy can be represented as a **directed acyclic graph** $H = (U, A)$ such that each triangle T of a planar subdivision S_i in the sequence is represented by a vertex u_T with outgoing arcs into the vertices representing the triangles of S_{i-1} that intersect T. It follows that, once we have a triangle T of a planar subdivision in the sequence, we can find all the triangles that intersect T from the previous subdivision, in $O(1)$ sequential time.

Before we show how to construct a subdivision hierarchy associated with S, we consider the processing of a query on this structure. Given a query point p, we start by locating p in a region T of the subdivision $S_{h(n)}$ containing three vertices. This step can clearly be done in $O(1)$ sequential time. Given T, there are at most d triangles of $S_{h(n)-1}$ having nonempty intersection with T. Hence, we can search through these regions to isolate the region of $S_{h(n)-1}$ containing p. We continue in the same fashion until we reach the initial subdivision $S_1 = S$. At this point, we can identify the triangle of S_1 containing p. Therefore, locating a point p in a triangulated planar subdivision can be done in $O(h(n))$ sequential time, once a subdivision hierarchy of height $h(n)$ is determined.

We now turn to the problem of constructing a subdivision hierarchy.

9.3.2 CONSTRUCTION OF A SUBDIVISION HIERARCHY

A key assumption of a subdivision hierarchy is that a region of S_{i+1} intersects at most a constant number d of the regions of S_i. We now develop an algorithm that refines S_i into S_{i+1} while maintaining this property for $d = 6$.

Consider a subdivision S_i and an *internal* vertex v of S_i of degree $t \leq 6$. If we remove v and the edges incident on v, we obtain a planar subdivision in which all regions are triangles except a region R that used to contain v. The boundary of R consists of a simple polygon containing t segments, and hence R can be retriangulated in $O(1)$ sequential time using a brute-force algorithm. Each of the resulting new triangles can intersect at most t regions in S_i—namely, the regions that used to lie in R. This process can be applied concurrently to all low-degree vertices, as long as they are not adjacent. Hence, all low-degree vertices of an independent set X can be removed concurrently such that each of the new triangles intersects at most six of the triangles in S_i.

The height of the hierarchy depends on the rate at which the number of vertices is decreasing. The sizes of two consecutive subdivisions S_i and S_{i+1} are related by $|S_{i+1}| = |S_i| - |X|$, where X is the independent set used to generate S_{i+1} from S_i. Hence, if the size of X is a constant fraction of $|S_i|$, the sizes of the subdivisions will decrease as a geometric series and thus the height of the hierarchy will be logarithmic.

We have already seen how to construct an independent set of low-degree vertices in $O(1)$ time, using a linear number of operations (Algorithm 9.2). The size of such an independent set is a constant fraction of the size of its subdivision with high probability (Theorem 9.3). Using such an independent set, we obtain that $|S_{i+1}| \leq c|S_i|$ with high probability for some constant $c > 0$, indicating that the height of the hierarchy is logarithmic with high probability.

We are now ready to state our algorithm for constructing a subdivision hierarchy.

ALGORITHM 9.3
(Subdivision Hierarchy)

Input: *A triangulated planar subdivision S represented by its edge lists such that the edges on the list of a vertex v are arranged in a counterclockwise ordering as they appear around v in S.*
Output: *A hierarchy of subdivisions of height $O(\log n)$ with high probability.*

begin
1. *Set* $i := 1, S_1 := S$
2. **while** (S_i *has more than three vertices*) **do**
   - *2.1. Determine a fractional independent set X of S_i by using Algorithm 9.2.*
   - *2.2.* **for** *each internal vertex* $v \in X$ **pardo**
     - *Remove v and its incident edges, and retriangulate the resulting simple polygon. Determine, for each new triangle, the old triangles it intersects.*

2.3. Set $i := i + 1$, *and let* S_i *be the resulting triangulated planar subdivision.*

end

Theorem 9.4: *Algorithm 9.3 constructs a subdivision hierarchy* $\{S_i\}_{i=1}^{h(n)}$ *associated with the triangulated subdivision* S *such that, with high probability,* $\sum_{i=1}^{h(n)} |S_i| = O(n)$ *and* $h(n) = O(\log n)$. *The algorithm runs in* $O(\log n)$ *time using* $O(n)$ *operations, with high probability.*

Proof: We start by establishing the correctness of the algorithm. By the discussion preceding the statement of the algorithm, each region T of a subdivision S_{i+1} generated during the ith iteration of the **while** loop (step 2) intersects at most six triangles of S_i, since the degree of each vertex in X (generated by Algorithm 9.2) is at most 6.

Let n_i be the number of vertices in S_i, where $1 \leq i \leq h(n)$. We know, from Theorem 9.3, that the size of the independent set X_i generated by Algorithm 9.2 satisfies $Pr\{|X_i| \leq \alpha n_i\} \leq e^{-\beta n_i}$, for some positive constants $\alpha < 1$ and β. This bound implies that $n_{i+1} \leq (1 - \alpha)n_i$ with high probability, for any given index i, unless n_i is very small. However, it is not clear that this fact holds for all indices $1 \leq i \leq h(n)$ simultaneously. We now proceed to prove that it essentially does hold.

Let k be the largest index such that $n_k = O(\log n)$ and $n_k \geq (\delta/\beta) \log n$, where δ is any positive constant. Clearly, such an index always exists. Consider the event E for which there exists an index i between 1 and k such that the independent set X_i found during the ith iteration satisfies $|X_i| \leq \alpha n_i$. By Boole's inequality (Eq. 9.1), the probability of this event satisfies $Pr\{E\} \leq \sum_{1 \leq i \leq k} Pr\{|X_i| < \alpha n_i\} \leq ke^{-\beta n_k}$, and the last inequality follows, since $n_k \leq n_i$ for $1 \leq i \leq k$. But $ke^{-\beta n_k} < ke^{-\delta \log n}$, since $n_k \geq (\delta/\beta) \log n$; hence, $Pr\{E\} < n^{-c\delta}$, for some positive constant c, and any positive constant δ. Therefore, we have that $|S_{i+1}| \leq (1 - \alpha)|S_i|$, for all i between 1 and k, with high probability. It follows that $k = O(\log n)$ with high probability.

Since $n_k = \Theta(\log n)$, the height can increase by at most a logarithmic factor (in n) after the kth iteration. Therefore, the height $h(n)$ of the subdivision hierarchy is $O(\log n)$ with high probability.

We now discuss the complexity bounds of the ith iteration of the **while** loop. Step 2.1 requires $O(1)$ time, using $O(n_i)$ operations. As for step 2.2, we need to create a representation of the subdivision S_{i+1} generated during the ith iteration from a representation of S_i. This step involves removing and inserting a number of edges from and into some edge lists of S_i.

Recall that a triangulated planar subdivision is represented by a set of edge lists such that the list corresponding to a vertex v consists of the edges arranged in the counterclockwise ordering of the edges around v in S_i. Hence,

during the process of removing the edges in step 2.2, no two edges can appear consecutively on a list because X is independent. It follows that each edge can be removed in $O(1)$ sequential time. Similarly, no two edges that have to be inserted on a list (because of retriangulation) appear consecutively, and each such edge can be inserted in $O(1)$ sequential time. Thus, the lists can be updated in $O(1)$ time, using $O(n_i)$ operations.

The tasks involved in forming the section of the directed acyclic graph connecting the triangles of S_{i+1} to those of S_i can be performed easily in $O(1)$ time, using $O(n_i)$ operations.

It follows that each iteration of the **while** loop can be implemented in $O(1)$ time, using $O(n_i)$ operations. Since $n_{i+1} \leq (1 - \alpha)n_i$ for all $1 \leq i \leq k$ with high probability, it follows that $\sum_i n_i = O(n)$ and hence the total number of operations is $O(n)$ with high probability. □

PRAM Model: Algorithm 9.3 requires concurrent-read capability in the loop defined at step 2. No concurrent write instructions are used. Hence, the algorithm runs on the CREW PRAM model. □

The next section introduces randomization techniques for handling the string-matching problem, which we discussed in detail in Chapter 7.

9.4 Pattern Matching

We have developed, in Chapter 7, a deterministic parallel algorithm to identify all the positions where a pattern string can occur in a text string (Sections 7.3 and 7.4). The algorithm proceeded in two phases. The first phase consisted of a *pattern analysis*, which generates a table related to the periodic structure of the pattern. The resulting optimal algorithm (Section 7.4.2) involves a number of complex details that make the algorithm unattractive for practical use.

The second phase (*text analysis*) consists of using the table generated during the pattern-analysis phase to determine all the positions where the pattern occurs. An optimal logarithmic-time algorithm was derived to handle the text analysis whenever the pattern is nonperiodic (Algorithm 7.2). This algorithm was complemented by another algorithm that reduces the periodic case to the nonperiodic case (Algorithm 7.3). As in the case of the pattern-analysis algorithm, there are various intricate details that make the overall text-analysis algorithm rather complex.

In this section, we address the string-matching problem and its two-dimensional generalization using randomization techniques. We describe

simple optimal logarithmic-time parallel algorithms that will succeed in identifying all the occurrences of the pattern, but that, with a very small probability, may report false occurrences.

The basic strategy used is similar to **hashing.** Briefly, hashing is a method to set up a table whose elements come from a large universe U. The table is determined dynamically during the processing of a task. The size of the table is usually much smaller than the size of U. The method consists of an easy-to-compute **hash function** that maps U into a small range of integers—say, the range $[1, 2, \ldots, r]$. During the processing of a task, the elements $u \in U$ that arise are mapped into the locations specified by $h(u)$. The hash function h is likely to distribute the elements evenly among the locations $1, 2, \ldots, r$, if h and r are chosen appropriately.

To handle the string-matching problem where the pattern is of length m, we map arbitrary strings of length m into integers of $O(\log m)$ bits, such that two distinct strings are mapped into the same integer with a very small probability. The value of the integer corresponding to a string X can be viewed as a "fingerprint" of X that can be compared to the fingerprint of the pattern. If the two fingerprints match, we report an occurrence of the pattern. We show that the algorithm reports false occurrences with a very small probability. Our analysis draws on two well-known facts from number theory.

For the remainder of this section, we assume that our strings are taken over the alphabet $\Sigma = \{0, 1\}$. The same techniques can be used to handle the case when the size of the alphabet is fixed.

9.4.1 FINGERPRINT FUNCTIONS

As we mentioned, our basic strategy amounts to mapping strings into small integers. Before defining our mapping, we describe the general framework and the properties needed to obtain the optimal randomized string-matching algorithms.

Let T be a text string of length n, and let P be a pattern string of length $m \leq n$. Our goal is to identify all the locations of T where P occurs. Consider the set B of all the substrings of T of length m. Notice that there are $n - m + 1$ such substrings, each starting at a location i such that $1 \leq i \leq n - m + 1$. Our problem can be reformulated as that of identifying the elements of B that are identical to P. The main problem lies in the fact that the comparison of a string from B and the pattern string P requires $\Theta(m)$ operations, and hence each such comparison is costly. We shall use the following alternative approach.

Let $\mathcal{F} = \{f_p\}_{p \in S}$ be a collection of functions such that f_p maps a string of length m into a domain D. Then, we require that the set $\mathcal{F}$ satisfies the following three properties:

Property 9.1: For any string $X \in B, f_p(X)$ consists of $O(\log m)$ bits for each $p \in S$. □

Property 9.2: The probability that a randomly chosen $f_p \in \mathcal{F}$ maps two unequal strings X and Y into the same element of D is very small. □

Property 9.3: For each $p \in S, f_p(X)$ should be easily computable in parallel for all the strings in B. □

The implications of Properties 9.1, 9.2, and 9.3 for our pattern-matching problems should be clear. Property 9.1 implies that $f_p(X)$ and $f_p(P)$ can be compared in $O(1)$ sequential time, since a pointer to a location in P requires $\Theta(\log m)$ bits. We shall call $f_p(X)$ a **fingerprint** of the string X. Property 9.2 states that, if the fingerprints of two strings X and Y are equal, then $X = Y$ with high probability. The interpretation of Property 9.3 is related to the parallel implementation of the algorithm that makes use of the set $\mathcal{F}$. Since our deterministic parallel string-matching algorithm achieves $O(\log n)$ time, using $O(n)$ operations, we wish to compute the fingerprints of the strings in B within the same complexity bounds (or faster).

We next identify a class $\mathcal{F}$ of functions that will satisfy Properties 9.1, 9.2, and 9.3.

A Class of Fingerprint Functions. We start by considering the following function f from the set of strings over $\{0, 1\}$ to the set of 2×2 matrices over the ring of integers Z. This function will satisfy Property 9.2, but will not satisfy Properties 9.1 or 9.3. We shall refine it later such that all the three properties are satisfied.

Let $f(0)$ and $f(1)$ be defined as follows:

$$f(0) = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \quad \text{and} \quad f(1) = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}.$$

For any two strings X and Y, $f(XY)$ is defined to be the product of the matrices $f(X)$ and $f(Y)$ over the ring Z. Hence, f is a *homomorphism* from the set of strings into the set of 2×2 matrices over Z. That is, f preserves the structures of the two domains by transforming a concatenation operation into matrix product. It is this very property that will allow the efficient computation of f on the required set of substrings.

EXAMPLE 9.9:

Consider the string $X = 1011$. Then, $f(X) = f(1)f(0)f(1)f(1)$, and hence

$$f(X) = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 5 \\ 1 & 3 \end{pmatrix}.$$ □

We are ready to state the main properties of the function f that will be needed later. The proof of the following lemma is left to Exercise 9.14.

Lemma 9.3: *The function f is one to one such that the determinant of $f(X)$ is equal to 1 for any string X. If X is of length m, then each entry of $f(X)$ is an integer no larger than F_{m+1}, where F_{m+1} is the $(m+1)$st Fibonnacci number (that is, $F_1 = F_2 = 1$, and $F_{m+1} = F_m + F_{m-1}$).* □

Clearly, the function f is not satisfactory for our purposes, since an entry of $f(X)$ could be as large as $F_{m+1} \approx \frac{\phi^{m+1}}{\sqrt{5}}$, where ϕ is the golden ratio whose value is $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618 \cdots$, and hence property [P1] is not satisfied. We now show how to modify f such that a substantial compaction will result.

Let p be a prime in the range $[1, 2, \ldots, M]$, where M is an integer to be specified later, and let Z_p be the ring of integers modulo p. For each such p, we define $f_p(X)$ to be $f(X)$ modulo p; that is, $f_p(X)$ is a 2×2 matrix such that its (i, j) entry is equal to the (i, j) entry of $f(X)$ reduced modulo p.

EXAMPLE 9.10:

Consider the string $X = (1011)^4$ of length 16. It is straightforward to verify that $f(X)$ is given by

$$f(X) = \begin{pmatrix} 206 & 575 \\ 115 & 321 \end{pmatrix}.$$

For the prime $p = 7$, we get

$$f_7(X) = \begin{pmatrix} 3 & 1 \\ 3 & 6 \end{pmatrix}.$$ □

We define $\mathcal{F}$ to be the class of functions f_p, where p is a prime in $[1, 2, \ldots, M]$. Clearly f_p maps a string of length m into a set of four integers, each of length $O(\log m)$ bits, whenever M is polynomial in m. Hence, f_p satisfies Property 9.1 whenever M is bounded by a polynomial in m. Before addressing whether Property 9.2 is satisfied, we need to state two well-known facts from number theory.

Two Facts from Number Theory. For any integer m, let $\pi(m)$ be the number of primes that are less than or equal to m. Then, we have the following facts, whose proofs can be found in the references cited at the end of this chapter (see bibliographic notes).

Lemma 9.4: *For any integer $m \geq 17$, the following inequalities hold:*

$$\frac{m}{\ln m} \leq \pi(m) \leq 1.2551 \frac{m}{\ln m}.$$ □

Lemma 9.5: *Given an integer $u \leq 2^m$, the number of distinct prime divisors of u is smaller than $\pi(m)$, where $m \geq 29$.* □

We now have the necessary tools to establish the validity of Property 9.2 for the class $\mathcal{F}$ of functions just introduced.

Probability of a False Match. Let X and Y be two strings, each of length m, and let $f_p \in \mathcal{F}$. A **false match** occurs when $X \neq Y$ and yet $f_p(X) = f_p(Y)$. Note that this property is relative to the particular fingerprint function f_p.

Theorem 9.5: *Let f_p be a randomly chosen function from the set $\mathcal{F} = \{f_p\}$, where p is a prime in the range $[1, 2, \ldots, M]$. Then, the probability of a false match between any two distinct strings of length m is $\leq \frac{\pi(\lceil 2.776m \rceil)}{\pi(M)}$, provided $m \geq 11$.*

Proof: Denote

$$f(X) = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix} \quad \text{and} \quad f(Y) = \begin{pmatrix} b_1 & b_2 \\ b_3 & b_4 \end{pmatrix}.$$

Since f is one-to-one and $X \neq Y$, there exists at least one index i such that $a_i \neq b_i$. Notice that, by Lemma 9.3, each of a_i and b_i is no larger than the Fibonnacci number $F_{m+1} \approx \phi^{m+1}/\sqrt{5}$, where $\phi = (1+\sqrt{5})/2$.

On the other hand, $f_p(X) = f_p(Y)$ implies that p divides the numbers $|a_i - b_i|$, for all $a_i \neq b_i$. Hence, the event $f_p(X) = f_p(Y)$, for $X \neq Y$, implies that the event p divides the number $u = \prod_{a_i \neq b_i} |a_i - b_i|$. Therefore, the probability of a false match is bounded by the probability that p divides u. However, by Lemma 9.3, $u \leq F_{m+1}^4 = 2^{4 \log F_m + 1}$. Using Lemma 9.5, the number of distinct prime divisors of u is $< \pi(\lceil 4 \log F_{m+1} \rceil)$. Since the number of distinct primes in the range $[1, 2, \ldots, M]$ is $\pi(M)$, we obtain

$$Pr\{\text{false match}\} \leq \frac{\pi(\lceil 4 \log F_{m+1} \rceil)}{\pi(M)} \leq \frac{\pi(\lceil 2.776m \rceil)}{\pi(M)},$$

since $\log \phi \approx .694$, and hence $\log F_{m+1} \approx .694m$. The condition $m \geq 11$ implies the condition $\lceil 2.776m \rceil \geq 29$. □

Remark 9.1: We can strengthen the bound of Theorem 9.5 slightly by observing that the probability of a false match is bounded by the probability that p divides $|a_j - b_j|$, for some j such that $a_j \neq b_j$. The latter probability is upper bounded by $\pi(\lceil \log F_{m+1} \rceil)/\pi(M) \leq \pi(\lceil .694m \rceil)/\pi(M)$. □

Corollary 9.1: *If the range of the primes is $[1, 2, \ldots, m^k]$, the probability of a false match between two strings of length m is $\leq \frac{3.48k}{m^{k-1}}$, for any given constant $k > 1$.*

Proof: We know from Lemma 9.4 that

$$\pi(\lceil 2.776m \rceil) \leq 1.2551 \frac{(2.776m)}{\ln(2.776m)} \approx \frac{3.48m}{\ln(2.776m)},$$

and

$$\pi(m^k) \geq \frac{m^k}{\ln m^k} = \frac{m^k}{k \ln m}.$$

The result follows from these inequalities and Theorem 9.5. □

In our pattern-matching algorithms, we will be interested in bounding the probability of a false match among many pairs of strings of the same length. Precisely the same arguments will establish the following theorem and its corollary.

Theorem 9.6: *For a randomly chosen function $f_p \in \mathcal{F}$, where p is a prime in the range $[1, 2, \ldots, M]$, the probability of a false match occurring among t pairs of strings $\{(X_i, Y_i)\}_{1 \le i \le t}$, each of length m, is bounded by $\frac{\pi(\lceil 2.776mt \rceil)}{\pi(M)}$, provided $mt \ge 11$.* □

Corollary 9.2: *Let k be any positive integer constant, and let $M = mt^k$. For this choice of M, the probability of a false match among t pairs of strings, each of length m, is bounded by $O\left(\frac{1}{t^{k-1}}\right)$.* □

A prime p in the range $[1, 2, \ldots, mt^k]$ requires $O(\log m + \log t)$ bits, for fixed k. For t pairs of such strings, pointers to specific locations of these strings will require the same order of bits. Hence, we can assume that the fingerprints of two such strings can be compared in $O(1)$ sequential time. This value should be contrasted with the fact that the comparison of the two strings will require $\Theta(m)$ operations.

We now address the validity of Property 9.3 relative to the class $\mathcal{F}$ of fingerprint functions.

Computation of Fingerprints. The key property of the class $\mathcal{F} = \{f_p\}$ of fingerprint functions that makes the latter computationally attractive is that each f_p is a *homomorphism*, that is, $f_p(X_1X_2) = f_p(X_1)f_p(X_2)$, for any two strings X_1 and X_2. We now provide an efficient parallel algorithm for computing the fingerprints of all the substrings of the text string T defined by $T(i{:}i + m - 1)$, where $1 \le i \le n - m + 1$ and m is the length of the pattern string P.

For each $1 \le i \le n$, let $N_i = f_p(T(1{:}i))$. Clearly, $N_i = f_p(T(1))f_p(T(2)) \cdots f_p(T(i))$. Since matrix multiplication is associative, this computation is a prefix-sums computation. Each matrix is of size 2×2; hence, the product of two such matrices can be found in $O(1)$ sequential time. Therefore, all the N_i's can be computed in $O(\log n)$ time, using a total of $O(n)$ operations.

Define

$$g_p(0) = f_p(0)^{-1} = \begin{pmatrix} 1 & 0 \\ p-1 & 1 \end{pmatrix},$$

and

$$g_p(1) = f_p(1)^{-1} = \begin{pmatrix} 1 & p-1 \\ 0 & 1 \end{pmatrix}.$$

Then, the product $R_i = g_p(T(i))g_p(T(i-1)) \cdots g_p(T(1))$ is a prefix-sums computation that can be done in $O(\log n)$ time, using $O(n)$ operations, for $1 \leq i \leq n$.

It is easy to check that $f_p(T(i{:}i+m-1)) = R_{i-1}N_{i+m-1}$. Hence, each such matrix can be computed in $O(1)$ sequential time, once all the R_i's and N_j's have been computed. Therefore, all the fingerprints can be computed in $O(\log n)$ time, using a total of $O(n)$ operations.

We therefore have the following lemma.

Lemma 9.6: *The fingerprints of all substrings of length m of a text string T of length $n \geq m$ can be computed in $O(\log n)$ time, using a total of $O(n)$ operations.* □

9.4.2 STRING MATCHING

We shall use the fingerprint functions defined in Section 9.4.1 to develop a randomized string-matching algorithm. Let $T(1{:}n)$ and $P(1{:}m)$ be the text and the pattern strings, respectively, over the alphabet $\Sigma = \{0, 1\}$, where $m \leq n$. Our goal is to identify all the locations of T where a matching of P occurs. We shall develop a randomized algorithm of the Monte Carlo type. Development of a Las Vegas algorithm is left to Exercise 9.16.

We have already sketched the basic strategy of a randomized string-matching algorithm of the Monte Carlo type. We compute the fingerprint of the pattern P and of the substrings $T(i{:}i+m-1)$, where $1 \leq i \leq n-m+1$. We then declare an occurrence of P in the text T at position i whenever the fingerprints of P and $T(i{:}i+m-1)$ are equal. The details are given next.

ALGORITHM 9.4
(Monte Carlo String Matching)
Input: *Two arrays $T(1{:}n)$ and $P(1{:}m)$ representing the text and the pattern strings, respectively, and an integer M.*
Output: *The array MATCH indicating all the positions where the pattern occurs in the text.*
begin
1. **for** $1 \leq i \leq n-m+1$ **pardo**
   Set $MATCH(i) := 0$
2. *Choose a random prime in the range* $[1, 2, \ldots, M]$, *and compute* $f_p(P)$.
3. **for** $1 \leq i \leq n-m+1$ **pardo**
   Set $L_i := f_p(T(i{:}i+m-1))$
4. **for** $1 \leq i \leq n-m+1$ **pardo**
   if $(L_i = f_p(P))$ **then** *Set* $MATCH(i) := 1$

end

Theorem 9.7: *Algorithm 9.4 reports all the occurrences of the pattern P in the text T, and may report false occurrences. The probability that there exists some i such that MATCH(i) = 1 and P does not occur in T at location i is* $O\left(\frac{1}{n^k}\right)$, *if we take* $M = mn^{k+1}$, *where k is any constant greater than or equal to* 1. *This algorithm can be implemented in* $O(\log n)$ *time, using a total of* $O(n)$ *operations.*

Proof: If the pattern P occurs at location i in the text T, then $f_p(P) = f_p(T(i{:}i + m - 1))$ for any prime p. Hence, each such occurrence will be reported. The probability of reporting a false occurrence follows immediately from Theorem 9.6 and Corollary 9.2. The complexity bounds are straightforward in view of Lemma 9.6. □

PRAM Model: Algorithm 9.4 can be implemented within the stated bounds without any simultaneous memory access, since the only nontrivial operations required are prefix-sums computations and broadcasting of the values of p and $f_p(P)$. Hence, Algorithm 9.4 can be implemented on the randomized EREW PRAM. □

Its simplicity, and the fact that it requires no preprocessing of the pattern, make Algorithm 9.4 attractive for practical use. In addition, if an upper bound on the size of the text is known, the random prime p can be generated in a preprocessing step.

EXAMPLE 9.11:

Let $m = 2^8$, and $n = 2^{12}$. Choose $M = mn^2 = 2^{32}$. Hence, each string of length 256 is mapped into an integer requiring 32 bits. Using the bounds shown previously, it is easy to verify that the probability of a false match is $< 2^{-9}$. □

We can decrease the error probability substantially by choosing several random primes, instead of just one. The proof of the corresponding bound is left to Exercise 9.17. On the other hand, we can eliminate any possibility of errors by modifying the algorithm as indicated in Exercise 9.16.

We can now extend our techniques to solve a two-dimensional pattern-matching problem.

9.4.3 TWO-DIMENSIONAL ARRAY MATCHING

Up to this point, we have restricted our discussions to strings. A more interesting class of problems is concerned with finding all the occurrences of a multidimensional pattern of arbitrary shape in a multidimensional array, where the elements are drawn from a certain alphabet. Several of our techniques can be extended to multidimensional matching problems. In this section, we consider the two-dimensional case where the pattern is also an array.

Let $T(n, n)$ and $P(m, m)$ be the text and the pattern arrays over the alphabet $\Sigma = \{0, 1\}$. The desired output is the array $MATCH$ such that $MATCH(i,j) = 1$ if and only if the pattern P occurs in T as a subarray whose top-left corner is located at (i,j). More precisely, $T(i + r - 1, j + s - 1) = P(r, s)$, where $1 \leq r, s \leq m$ and $1 \leq i, j \leq n - m + 1$ (see Fig. 9.2 for an illustration).

As before, we use the class of functions $\mathcal{F} = \{f_p \mid p \text{ a prime} \leq M\}$ to compute the fingerprints of all the appropriate $m \times m$ subarrays of T. Such a subarray is viewed as a single string of length m^2 consisting of the first row of the subarray followed by the second row, then the third row, and so on.

EXAMPLE 9.12:

Let the 4×4 pattern P be given by

$$P = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix}.$$

Then, a fingerprint of P can be $f_7(X)$, where X is the string 1011001111001011. It is straightforward to verify that the corresponding fingerprint is given by

$\begin{pmatrix} 2 & 0 \\ 1 & 4 \end{pmatrix}$.

□

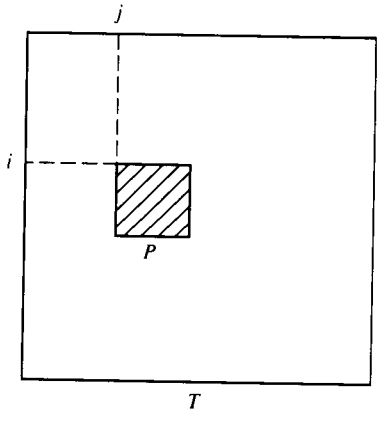


FIGURE 9.2
Two-dimensional array matching. The pattern P occurs at location (i, j) if the overlapping portions are identical.

Let $L_{i,j}$ be the fingerprint of the subarray of T whose top-left corner is located at position (i, j). There are $(n - m + 1)^2$ such subarrays. As in Algorithm 9.4, we can check, for all pairs (i, j) such that $1 \leq i, j \leq n - m + 1$, whether $L_{i,j} = f_p(P)$, for a randomly chosen prime $p \leq M$. We then set $MATCH(i, j) := 1$ whenever the equality holds. The probability of a false match is $\leq \frac{\pi(\lceil 2.776m^2(n-m+1)^2 \rceil)}{\pi(M)}$, by Theorem 9.6. Taking $M = m^2 n^{k+2}$, we obtain that the error probability is $O\left(\frac{1}{n^k}\right)$, for any constant $k > 0$.

The complexity bounds of the resulting randomized algorithm remain to be determined. The only details that need to be specified are those concerning the computation of the required fingerprints, which is addressed next.

Computation of the Fingerprints. The fingerprints $\{L_{i,j}\}$ required for all the subarrays of T of size $m \times m$ can be computed in two main steps, as follows:

- *Step 1:* For each row i, where $1 \leq i \leq n$, compute $S_{i,j}$, the fingerprint of the substring in row i of length m starting at location (i, j), for all $1 \leq j \leq n - m + 1$. For each row, this step requires $O(\log n)$ time, using a total of $O(n)$ operations (Lemma 9.6). Therefore, all the matrices $S_{i,j}$ can be computed in $O(\log n)$ time, using a total of $O(n^2)$ operations.
- *Step 2:* For each column j, where $1 \leq j \leq n - m + 1$, compute $L_{i,j} = S_{i,j}S_{i+1,j} \cdots S_{i+m-1,j}$, for all $1 \leq i \leq n - m + 1$. Again, this step can be done with the algorithm referred to in Lemma 9.6. Therefore, this step takes $O(\log n)$ time, using a total of $O(n(n - m + 1))$ operations.

Putting the Pieces Together. The following theorem provides a summary regarding the randomized algorithm developed for the two-dimensional array matching.

Theorem 9.8: *The two-dimensional array-matching problem can be solved by a randomized algorithm in $O(\log n)$ time, using a linear number of operations, where the text array is of size $n \times n$. The probability that the algorithm reports a false matching can be bounded by $O\left(\frac{1}{n^k}\right)$, for any constant k.* □

The algorithm described in this section can be extended to handle the following two more general problems: (1) multidimensional array matching, and (2) the case where patterns are of irregular shapes. These extensions will be left as Exercises 9.18 and 9.19.

We consider next randomized algorithms for a class of algebraic problems.

9.5 Verification of Polynomial Identities

Let $p(x_1, \ldots, x_n)$ be a polynomial in the variables $x_1, x_2, \ldots, x_n$ over an arbitrary field F. Then, $p(x_1, \ldots, x_n)$ can always be expressed as a sum of multinomials $a_{i_1,i_2,\ldots,i_n} x_1^{i_1} x_2^{i_2} \ldots x_n^{i_n}$, where $a_{i_1,i_2,\ldots,i_n} \in F$. The *degree* of $p(x_1, \ldots, x_n)$, denoted by $\deg(p)$, is equal to $\max\{i_1 + i_2 + \ldots + i_n\}$, where the maximum is over all multinomials of $p(x_1, \ldots, x_n)$ such that $a_{i_1,i_2,\ldots,i_n} \neq 0$.

EXAMPLE 9.13:

Let X be the following matrix

$$\begin{bmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \\ x_7 & x_8 & x_9 \end{bmatrix},$$

where the x_i's are indeterminates. Consider the polynomial $p(x_1, \ldots, x_9) = \det(X) = x_1x_5x_9 - x_1x_6x_8 - x_2x_4x_9 + x_2x_6x_7 + x_3x_4x_8 - x_3x_5x_7$. Clearly, all the multinomials have the same degree 3, and hence the degree of $p(x_1, \ldots, x_9)$ is 3. □

We consider, in this section, the problem of verifying whether a multivariate polynomial is identically zero. The straightforward method of expanding the polynomial into the sum of its multinomials and checking whether all the coefficients are equal to zero requires, in general, an excessive number of operations. We explore a different approach based on evaluating the polynomial at a point (which is an n-dimensional vector, where n is the number of variables) chosen randomly over a finite set I. We show that, if the polynomial is not identically zero, the value of the polynomial will be different from zero with probability at least 1/2, whenever I is chosen large enough. The experiment can be repeated to reduce the error probability below any small constant.

Before proceeding, we give two examples, one from graph theory and the other from matrix computations, where the solution to each of the related problems reduces to testing whether a certain polynomial is identically zero.

EXAMPLE 9.14: (Perfect Matching)

Let $G = (V, E)$ be a graph whose vertex set V is denoted by $\{1, 2, \ldots, n\}$. A **perfect matching** in G is a set M of edges such that each vertex of G is incident on exactly one edge of M. This problem will be discussed in detail in Section 9.7. We mention here only that the graph G has a perfect matching if and only if the determinant of a matrix dependent on $|E|$ variables is not identically

zero. Expanding the determinant of such a matrix will be an expensive computational process. However, if we substitute constant values for the variables, the computation of the determinant over a field is substantially easier. □

EXAMPLE 9.15:

Let A be an $n \times n$ matrix over a field F. We are interested in finding out whether or not the matrices $I, A, A^2, \cdots, A^{n-1}$ are linearly independent. Note that, by the Cayley-Hamilton theorem (Theorem 8.15), the matrices $I, A, A^2, \cdots, A^n$ are always linearly dependent. The straightforward method consists of computing the matrix powers A^i, for $2 \leq i \leq n-1$, and then determining whether there exist $\alpha_0, \alpha_1, \ldots, \alpha_{n-1}$ such that $\sum_{i=0}^{n-1} \alpha_i A^i = 0$, where not all the α_i's are zeros. Another computationally more efficient method is the following.

Let x be an n-dimensional vector of indeterminates. Then, the matrices $I, A, A^2, \cdots, A^{n-1}$ are linearly independent if and only if the Krylov matrix $[x\ Ax\ A^2x \cdots A^{n-1}x]$ (see Example 8.3) is nonsingular over F. This characterization is in turn equivalent to testing whether the determinant of the Krylov matrix is identically zero. Hence, the problem reduces to testing whether a polynomial in several variables is identically zero. □

We next introduce the basic technique more formally.

9.5.1 THE BASIC TECHNIQUE

We are ready to state and prove the main fact that will be the basis of our randomized algorithm for testing whether a polynomial is identically zero.

Theorem 9.9: *Let $p(x_1, \ldots, x_n)$ be a polynomial in the variables $x_1, x_2, \ldots, x_n$ over a field F such that $p(x_1, \ldots, x_n)$ is not identically zero. Let I be any finite subset of F. Then, the number of elements in I^n which are zeros of $p(x_1, \ldots, x_n)$ is at most $|I|^{n-1}\deg(p)$.*

Proof: The proof is by induction on n.

The base case is when $n = 1$. In this case, we have to show that the number of zeros in I of a polynomial $p(x)$ of a single variable is at most $\deg(p)$. But this is a well-known fundamental result in algebra; hence, the theorem holds for $n = 1$.

Assume that the induction hypothesis holds for all polynomials with at most $n-1$ variables. Let δ be the degree of x_1 in $p(x_1, \ldots, x_n)$. Clearly, we have that $p(x_1, \ldots, x_n) = x_1^{\delta} q(x_2, \ldots, x_n) + r(x_1, x_2, \ldots, x_n)$, for some polynomials q and r. Let $(\alpha_1, \alpha_2, \ldots, \alpha_n) \in I^n$ be a zero of $p(x_1, \ldots, x_n)$.

Then, either $q(\alpha_2, \ldots, \alpha_n)$ is equal to zero or it is different from zero. If $q(\alpha_2, \ldots, \alpha_n) = 0$, then $p(\alpha_1, \alpha_2, \ldots, \alpha_n)$ might be equal to zero for all $\alpha_1 \in I$ (for example, in the case when r is identically zero). The total number of such zeros is then at most $|I|.(|I|^{n-2}\deg(q))$, where the term between parentheses follows from the induction hypothesis.

On the other hand, if $q(\alpha_2, \ldots, \alpha_n) \neq 0$, the corresponding total number of such zeros of p is at most $|I|^{n-1}\delta$, since p is of degree δ in x_1. Therefore, the total number of zeros in I^n is $\leq |I|^{n-1}(\delta + \deg(q)) \leq |I|^{n-1}\deg(p)$. □

We now consider a simple polynomial, and determine the exact number of its roots in an interval.

EXAMPLE 9.16:

Let F be an infinite field, and let I be a finite subset containing 0. Consider the polynomial $p(x_1, \ldots, x_n) = x_1x_2 \cdots x_n$. The number r of the roots of $p(x_1, \ldots, x_n)$ is equal to the number of n-tuples over I each with at least one 0. Hence, $r = |I|^n - (|I| - 1)^n$. Since

$$(|I| - 1)^n = \sum_{i=0}^{n}(-1)^i \binom{n}{i}|I|^{n-i},$$

we get that

$$r = \sum_{i=1}^{n}(-1)^{i+1}\binom{n}{i}|I|^{n-i} \leq n|I|^{n-1}.$$ □

The relevance of Theorem 9.9 in designing randomized algorithms for the verification of polynomial identities is stated more clearly in the next corollary.

Corollary 9.3: *Let $p(x_1, \ldots, x_n) \neq 0$ and I be as in Theorem 9.9. The probability that a random tuple $(\alpha_1, \ldots, \alpha_n) \in I^n$ is a zero of $p(x_1, \ldots, x_n)$ is $\leq \frac{\deg(p)}{|I|}$.* □

This corollary suggests the following randomized scheme for testing whether a polynomial $p(x_1, \ldots, x_n)$ is identically zero. We choose a finite subset of the domain of p that is of size at least 2 $\deg(p)$. We then select a random n-dimensional vector v over I. That is, each entry of v is chosen uniformly and independently from I. We evaluate the polynomial at v, and test whether the resulting value is equal to zero. If it is, we declare the polynomial p to be identically zero. Otherwise, the polynomial is clearly not identically zero. The error probability of such a scheme is at most 1/2. If the same experiment is performed k times (either sequentially or in parallel), the error probability reduces to at most $1/2^k$.

In Section 9.5.2, we use this technique for verifying matrix products.

9.5.2 VERIFICATION OF MATRIX PRODUCTS

As a simple application, we consider the following **verification** problem. Given three $n \times n$ matrices A, B and C, determine whether $AB = C$.

The obvious deterministic algorithm computes the matrix product AB, and then checks whether the corresponding entries are equal. Such a procedure requires $O(\log n)$ time, using a total of $O(M(n))$ operations, where $M(n)$ is the best known bound on the number of arithmetic operations required for $n \times n$ matrix multiplication. Next, we develop a simple randomized algorithm that can solve this problem in $O(\log n)$ time, using only $O(n^2)$ operations.

We start by establishing the following lemma.

Lemma 9.7: *Let u be an n-dimensional vector over a field F such that $u \neq [0, 0, \ldots, 0]^T$, and let v be an n-dimensional vector such that each entry of v is chosen randomly from the set $\{0, 1\}$. Then, the probability that $\sum_{i=1}^{n} u_i v_i = 0$ is less than or equal to $1/2$.*

Proof: Let $p(x_1, x_2, \ldots, x_n)$ be the polynomial defined by $p(x_1, \ldots, x_n) = \sum_{i=1}^{n} u_i x_i$, where the x_i's are indeterminates. Since u is not the zero vector, the polynomial p is not identically zero. By Theorem 9.9, the number of zeros of p over the set $I = \{0, 1\}$ is at most $|I|^{n-1} \deg(p) = 2^{n-1}$. Since the vector v is equally likely to be any of the vectors in I^n, the probability that v happens to be one of the zeros of p is at most $2^{n-1}/2^n = 1/2$. □

We are ready to prove our result concerning matrix-product verification.

Theorem 9.10: *Given three $n \times n$ matrices A, B, and C over an arbitrary field F, the problem of verifying whether $AB = C$ can be solved with a randomized parallel algorithm that runs in $O(\log n)$ time, using $O(n^2)$ operations, such that the probability of error is at most $1/2$.*

Proof: The algorithm consists of generating a random vector v whose entries are from $\{0, 1\}$. We apply the test $A(Bv) \stackrel{?}{=} Cv$. If the test fails, we declare that $AB \neq C$. Otherwise, we say that $AB = C$.

It is clear that, if $AB = C$, the algorithm will always generate the correct answer. However, the algorithm may generate an incorrect answer whenever $AB \neq C$. Therefore, let us estimate the probability of $A(Bv) = Cv$ in the case where $AB \neq C$.

Since $AB \neq C$, there exists at least one index j, where $1 \leq j \leq n$, such that the jth rows of AB and C are distinct. Let us denote such a row from AB by $u = [u_1, u_2, \ldots, u_n]$ and the corresponding row from C by $w = [w_1, w_2, \ldots, w_n]$. Hence we have that $e = u - w$ is a nonzero vector. The probability that $A(Bv) = Cv$ is less than or equal to the probability that

$e \cdot v = 0$, where $e \cdot v$ is the inner product of the two vectors. By Lemma 9.7, the probability of such an event is $\leq 1/2$. Therefore the probability that $A(Bv) = Cv$ when $AB \neq C$ is at most 1/2.

The complexity bounds are obvious. □

We can make the error probability very small by repeating the experiment several times.

Corollary 9.4: *Given three $n \times n$ matrices A, B, and C, we can test whether $AB = C$ with a randomized parallel algorithm that runs in $O(\log n)$ time, using $O(n^2k)$ operations, such that the error probability is at most $1/2^k$.* □

The technique described in this section turns out to be applicable to a wide variety of problems. In Section 9.7, we make a crucial use of this technique for solving the maximum matching problem. Exercises 9.20 through 9.24 will provide additional examples where this technique is applicable.

The next section considers randomized sorting algorithms.

9.6 Sorting

In this section, we consider the problem of sorting n elements drawn from a linearly ordered set. The sorting problem was considered in Chapter 4, where we described three deterministic parallel sorting algorithms—namely, the merge-sort algorithm (Algorithm 4.3), the pipelined merge-sort algorithm (Algorithm 4.4), and the bitonic-sorting algorithm (Section 4.4). The merge-sort algorithm (Algorithm 4.3) is a simple parallel implementation of the divide-and-conquer strategy that uses the optimal $O(\log \log n)$ time merging algorithm. The corresponding complexity bounds are $O(\log n \log \log n)$ time and $O(n \log n)$ operations. The implementation of this algorithm is rather complex, due to the required merging operations. The pipelined merge-sort algorithm (Algorithm 4.4) allows us to sort n elements in $O(\log n)$ time, using $O(n \log n)$ operations on the CREW PRAM (which can also be extended to the EREW PRAM). However, this algorithm is intricate, and the analysis of its complexity bounds is involved. The bitonic-sorting algorithm, described in Section 4.4, is the simplest algorithm to implement, but it runs in $\Theta(\log^2 n)$ time, using $\Theta(n \log^2 n)$ operations.

We introduce in this section the *random-sampling* technique that can be used to derive a simple optimal $O(\log n)$ time sorting algorithm.

9.6.1 RANDOM SAMPLING

Random sampling is a very powerful randomization technique that can be applied to a wide variety of problems. It can be used in the context of a divide-and-conquer or partitioning approach as follows. Let I be the input to our problem P. We choose a random sample S of I of a certain size. We process the elements of S such as to induce a partitioning of our initial problem into smaller-sized subproblems that can be solved in parallel. The success of this scheme depends crucially on the likelihood that the induced subproblems are of approximately equal sizes, and on the likelihood that the sum of the sizes of the subproblems is of approximately the same order as the size of the initial problem.

There are several ways to pick a sample of size r from a population of size n. We describe here two simple methods. Other sampling techniques are discussed in Exercises 9.32 and 9.33. **Sampling without replacement** can be viewed as the process of selecting one element at a time, each element having the same probability of being selected. Once an element is selected, it is removed from the population. **Sampling with replacement** can be viewed as the process of selecting one element at a time, except that the selected element is not removed from the population; that is, the same element can be selected several times, each time with the same probability as any other element in the population. Sampling with replacement can also be viewed as the process of rolling several dice; each has n faces numbered $1, 2, \ldots, n$, and each face is equally likely to show up.

For our parallel-algorithms setting, we shall adopt the sampling-with-replacement strategy in this section, whenever a sample of size r needs to be generated in $O(1)$ time. Put another way, we can assume that we have r processors, each of which has a random-number generator. During the sampling step, each processor draws a random number from the set $\{1, 2, \ldots, n\}$. Hence, we end up with a sample of size r, which may contain duplicate elements. As we shall see in Section 9.6.2, the fact that several processors may pick the same key will not affect the performance of the algorithm when n is large.

Another scheme that is suitable for parallel algorithms is to let each processor pick an element with probability r/n. The expected size of the sample is r. Showing that this scheme could be used instead of the sampling-with-replacement scheme is left as Exercise 9.32.

We are ready to outline the structure of a sorting scheme that uses the random-sampling technique.

9.6.2 RANDOMIZED QUICKSORT

The **quicksort** algorithm is an important practical sorting algorithm whose performance analysis is interesting. The basic algorithm can be viewed as an

application of the *partitioning strategy* outlined in Section 2.4. The elements of the input array are partitioned into a sequence of *buckets* $\{B_j\}$, where each element of bucket B_j is smaller than any element in the next bucket B_{j+1} (unless B_j is the last bucket). Once the input is partitioned in this fashion, the problem of sorting the given set of elements reduces to sorting the different buckets separately, and thus the buckets can be sorted concurrently. The main difficulty in carrying out this plan lies in developing an efficient scheme for partitioning the input such that the buckets are of approximately equal sizes.

The schemes that we use in this section for partitioning the input amount to *rearranging* the elements of the input array such that the elements of the bucket B_1 appear first, followed by the elements of B_2, and so on. Other schemes, based on **binary search trees,** do not require the elements to be rearranged after each partitioning step (see Exercises 9.25 and 9.26).

We shall start by describing a parallel version of a sequential randomized quicksort algorithm. We shall show that, with high probability, its running time is $O(\log^2 n)$ and its total number of operations is $O(n \log n)$. We shall later apply a $\sqrt{n}$ partitioning strategy to reduce the running time to $O(\log n)$ with high probability.

Without loss of generality, we shall assume for the remainder of this section that the elements to be sorted are distinct.

9.6.3 A PARALLEL RANDOMIZED QUICKSORT ALGORITHM

Let A be the array containing the n distinct elements to be sorted. We use a **two-way** partitioning strategy in this section. Hence our main task is to rearrange the elements of A into two subarrays B_1 and B_2, called **buckets,** such that each element of B_1 is smaller than any element of B_2, and $|B_1| \approx |B_2|$. We can identify a partitioning based on the following simple scheme.

We pick a random element $S(A)$ from the input array A; hence, each element of A is equally likely to be selected. The sample key $S(A)$ is called a **splitter.** We then compare $S(A)$ with every element of A, and mark the smaller elements with 1 and the larger elements with 0. Now, we move the smaller elements to the beginning of A, and the larger elements to the end of A, such that the element $S(A)$ is sandwiched between the two subsets. The same strategy can be applied to each bucket recursively until the size of each bucket is sufficiently small—say, 30 or less. We can then sort each small bucket by a simple sorting algorithm, such as selection sort.

We next describe the algorithm more formally.

ALGORITHM 9.5

(Randomized Quicksort)

Input: *An array A containing the n elements to be sorted.*

Output: *The array A in sorted order.*
begin
1. **if** $n \leq 30$ **then** *sort A using any sorting algorithm and* **exit.**
2. *Select a random element* $S(A)$ *of A.*
3. **for** $1 \leq i \leq n$ **pardo**
   $A(i) < S(A)$: *set mark*(i): $= 1$
   $A(i) > S(A)$: *set mark*(i): $= 0$
4. *Compact the elements of A marked* 1 *at the beginning of A, followed by* $S(A)$*, which is followed by the elements marked* 0*. Set k equal to the position of the element* $S(A)$.
5. *Recursively sort the subarrays* $A(1{:}k-1)$ *and* $A(k+1{:}n)$.

end

EXAMPLE 9.17:

For illustrative purposes, we assume that the recursion in Algorithm 9.5 bottoms out at $n = 1$ instead of $n = 30$. Consider the input array $A = (4, 16, -5, 7, 25, -8, 1, 3)$, shown in Fig. 9.3(a). During the first iteration, a sample key is selected—say, $A(4) = 7$—as the splitter, which results in the creation of two buckets, the first of size 5 and the second of size 2, as shown in Fig. 9.3(b). The second iteration of the algorithm causes two sample keys to be selected from the two buckets, as indicated by the two arrows in the figure. These keys induce the partitions shown in Fig. 9.3(c). The third iteration causes one splitter to be selected, resulting in two buckets, each of size 1, as shown in Fig. 9.3(d). The next recursive call causes the algorithm to terminate. □

The performance of Algorithm 9.5 depends on the number of recursive steps needed. We refer to each recursive step as a **partitioning step.** The number of partitioning steps depends on the rate at which the sizes of the intermediate buckets are reduced. In one pessimistic scenario, we may have very unbalanced partitions, causing the size of the largest bucket to decrease by a constant number of elements after each iteration. This situation occurs, for example, if the splitter in the largest bucket happens to be the second smallest element during each iteration. Therefore, in the worst case, the algorithm requires $\Theta(n)$ iterations, yielding a running time of $O(n \log n)$ (since each iteration can be performed in $O(\log n)$ time, as we show later), and a total number of $O(n^2)$ operations. However we show next that, with high probability, $O(\log n)$ iterations suffice to reduce the size of every bucket to 30 or less.

Theorem 9.11: *Algorithm 9.5 sorts a sequence of n elements in* $O(\log^2 n)$ *time, using* $O(n \log n)$ *operations, with high probability.*

Proof: We start by estimating the resources required for each iteration.

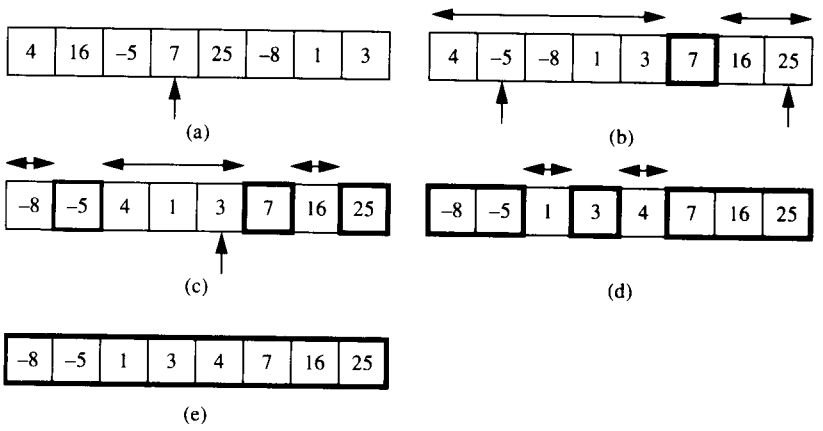


FIGURE 9.3
Randomized quicksort for Example 9.17. (a) An input to the randomized quicksort algorithm. The arrow points to the sample key selected during the first iteration. (b) The partitions induced by the element 7, as indicated. During the second iteration of the algorithm, two random keys are selected from the two buckets. These keys are indicated by the two arrows. (c) The partitioning induced by the two keys selected during the second iteration, and the corresponding partitions. (d) The partitions induced by the only sample key selected during the third iteration. (e) The sorted array generated during the fourth iteration.

Steps 1 and 2 take $O(1)$ sequential time. Step 3 requires $O(1)$ parallel time, using $O(n)$ operations. Step 4 involves a prefix-sums computation, and hence can be performed in $O(\log n)$ time, using $O(n)$ operations. It follows that each iteration of the algorithm can be completed in $O(\log n)$ time, using a linear number of operations. It remains to establish a bound on the number of iterations.

Our strategy will be as follows. For any specific element e, we show that the sizes of any two consecutive buckets containing e decrease by a constant factor with a certain probability (Claim1). We then show that, with probability $1 - O(n^{-7})$, the bucket containing e will be of size 30 or less after $O(\log n)$ iterations (Claim2). Using Boole's inequality (Eq. 9.1), we conclude that, with probability $1 - O(n^{-6})$, the bucket of every element is of size ≤ 30 after $O(\log n)$ iterations. We next show how to carry out this strategy.

Let e be an arbitrary element of our input array A. Let n_j be the size of the bucket containing e at the end of the jth partitioning step, where $j \geq 1$. We set $n_0 = n$. Then, the following claim holds:

Claim1: $Pr\{n_{j+1} \geq 7n_j/8\} \leq 1/4$, for any $j \geq 0$.

Proof of Claim1: An element a partitions the jth bucket into two buckets, one of which is of size at least $7n_j/8$ if and only if $rank(a{:}B_j) \leq n_j/8$ or $rank(a{:}B_j) \geq 7n_j/8$. The probability that a random element is among the smallest or largest $n_j/8$ elements of B_j is $\leq 1/4$; hence, Claim1 follows.

We fix the element e of A, and we consider the sizes of the buckets containing e during various partitioning steps. We call the jth partitioning step **successful** if $n_j < 7n_{j-1}/8$. Since $n_0 = n$, the size of the bucket containing e after k successful partitioning steps is smaller than $(7/8)^k n$, and hence e can participate in at most $c \log(n/30)$ successful partitioning steps, where $c = 1/\log(8/7)$. For the remainder of this proof, all logarithms are to the base 8/7, and hence the constant c is equal to 1.

Claim2: Among $20 \log n$ partitioning steps, the probability that an element e goes through $20 \log n - \log(n/30)$ unsuccessful partitioning steps is $O(n^{-7})$.

Proof: The random choices made at various partitioning steps are **independent**; hence, the events consisting of the partitioning steps being successful are independent. It follows that we can model these events as *Bernoulli trials*. Let X be a random variable denoting the number of unsuccessful partitioning steps among the $20 \log n$ steps. Then, we have

$$Pr\{X > 20 \log n - \log(n/30)\} \leq Pr\{X > 19 \log n\}$$
$$\leq \sum_{j > 19 \log n} \binom{20 \log n}{j} \left(\frac{1}{4}\right)^j \left(\frac{3}{4}\right)^{20 \log n - j}.$$

Given that

$$\binom{n}{k} \leq \left(\frac{en}{k}\right)^k$$

(as stated in Eq. 9.6), we obtain that our probability is

$$\leq \sum_{j > 19 \log n} \left(\frac{20e \log n}{j}\right)^j \left(\frac{1}{4}\right)^j = \sum_{j > 19 \log n} \left(\frac{5e \log n}{j}\right)^j$$
$$\leq \sum_{j > 19 \log n} \left(\frac{5e \log n}{19 \log n}\right)^j = \sum_{j > 19 \log n} \left(\frac{5e}{19}\right)^j = O(n^{-7}).$$

Therefore, Claim2 follows.

We have shown, so far, that, given an arbitrary element e of A, the probability that e goes through $20 \log n - \log(n/30)$ unsuccessful partitioning steps is $O(n^{-7})$. By Boole's inequality (Eq. 9.1), the probability that one or more elements of A go through so many unsuccessful steps is at most $O(n \times n^{-7}) = O(n^{-6})$; hence, with probability $1 - O(n^{-6})$, the algorithm terminates within $20 \log n$ iterations, and the proof of the theorem follows. □

PRAM Model: The random element $S(A)$ generated at step 2 of Algorithm 9.5 can be reproduced in n distinct locations in $O(\log n)$ time, using $O(n)$ operations, without simultaneous memory access. Hence, steps 3 and 4 do not require any concurrent memory accesses. Therefore, Algorithm 9.5 runs on the randomized EREW PRAM model. □

9.6.4 AN $O(\log n)$ TIME RANDOMIZED SORTING ALGORITHM

The partitioning strategy adopted in Algorithm 9.5 causes each bucket B_j arising at a partitioning step to be divided into two smaller buckets induced by sampling a key from B_j. The bulk of the work performed at each iteration involves the rearrangement of the elements of each bucket into two separate smaller buckets. We accomplished this task by using the prefix-sums algorithm for computing the new location of each element; hence, each partitioning step required logarithmic running time using a linear number of operations. Since, with high probability, the algorithm terminates in $O(\log n)$ iterations, the total running time was $O(\log^2 n)$ with high probability.

In this section, we shall generalize the two-way partitioning strategy by sampling $\sqrt{n}$ keys from each bucket instead of a single key. As before, the sample keys are called *splitters*. Each bucket will be partitioned into at most $\sqrt{n} + 1$ smaller buckets, and the algorithm will recursively sort the smaller buckets. The cost of each partitioning step remains logarithmic in the running time, but the overall time will also be logarithmic, as illustrated by the following intuitive argument.

Since we sample $\sqrt{n}$ keys from our input, we expect each resulting bucket to be of approximately size $\sqrt{n}$. At the ith iteration, we expect each bucket to be of approximately size $n^{1/2^i}$. All the buckets arising at any given iteration can be processed concurrently. Therefore, the execution time of the ith iteration is $O(\log n^{1/2^i}) = O((1/2^i) \log n)$, which implies that the total running time is given by $O(\sum_i (1/2^i) \log n) = O(\log n)$, as we have claimed.

Before presenting a rigorous analysis to support our intuitive argument, we describe our algorithm more formally.

ALGORITHM 9.6
($\sqrt{n}$ Sample Sort)
Input: *An array A containing n elements to be sorted.*
Output: *The array A containing the elements in sorted order.*
begin
1. If the number of elements to be sorted is $\leq$ 30, then use any sorting algorithm and exit. Otherwise go to step 2.

2. Choose a set S of $\sqrt{n}$ random samples drawn independently from the input A.
3. Sort the set S by comparing all pairs of elements of S and storing the outcomes in a $\sqrt{n} \times \sqrt{n}$ table T. Then, compute the rank of each element of S by using a prefix-sums algorithm on each row of T.
4. Rearrange the elements of A into buckets $\{B_i\}_{i=1}^{\sqrt{n}+1}$ such that the elements of B_i are those of A that are between the $(i-1)$st smallest and the ith smallest elements of the sample S. The 0th smallest element of S is defined to be $-\infty$ and the $(\sqrt{n}+1)$st smallest element is defined to be $+\infty$.
5. Recursively sort the elements in each bucket, for all buckets concurrently.
end

Theorem 9.12: *With high probability, Algorithm 9.6 terminates in $O(\log n)$ time, using $O(n \log n)$ operations.*

Proof: We assume that, when the size of a bucket is no larger than $16(\log n)^4$, the bitonic-sorting algorithm (Section 4.4) is used to sort the elements in the corresponding bucket. Hence, each such bucket will be sorted in $O((\log \log n)^2)$ time, using $O(\log^4 n(\log \log n)^2)$ operations. Since the sum of the sizes of the buckets that are sorted by the bitonic-sorting algorithm is n, this final processing adds $O((\log \log n)^2)$ to the running time and $O(n(\log \log n)^2)$ to the total number of operations. We now concentrate on analyzing the resources required to reduce the size of each bucket to $16(\log n)^4$ or less.

We start by establishing the complexity bounds required by each iteration of the algorithm.

Claim1: Each iteration of Algorithm 9.6 can be executed in $O(\log n)$ time, using $O(n \log n)$ operations on a bucket of size n.

Proof of Claim1: Step 1 takes $O(1)$ sequential time. According to our assumptions, we can generate $\sqrt{n}$ independent random numbers from $\{1, 2, \ldots, n\}$ in $O(1)$ time, using $O(\sqrt{n})$ operations. Hence, step 2 can be performed within these bounds. Constructing the table T required by step 3 can be done in $O(1)$ time, using $O(n)$ operations. Applying the prefix-sums algorithm to each row of T takes $O(\log n)$ time, using $O(n)$ operations.

The implementation of step 4 is more subtle and can be achieved in two stages. During the first stage, each element of A can be located between two consecutive elements of the sorted sample S, using the binary search algorithm. Hence, the bucket numbers of all the elements of A can be identified in $O(\log n)$ time, using a total of $O(n \log n)$ operations.

During the second stage, the elements of A are rearranged such that the elements whose bucket numbers are each equal to 1 appear first, followed by those whose bucket numbers are each equal to 2, and so on. This problem is equivalent to one of **integer sorting,** where the n integers to be sorted belong to the range $[1, 2, \ldots, \sqrt{n}]$. We can solve this problem by building a balanced binary tree on the elements of A such that each node v of the tree contains a set of linked lists, each linked list corresponds to the leaves with the same bucket number stored in the subtree rooted at v. Hence, generating the lists at each node consists of merging the lists of the same bucket number stored at the children nodes. The details are left to Exercise 9.28.

We thus see that the problem of rearranging the elements can be performed in $O(\log n)$ time, using $O(n \log n)$ operations, and the proof of Claim1 is complete.

Our next goal is to estimate the sizes of the buckets arising at successive partitioning steps of Algorithm 9.6.

Let e be an arbitrary element from our input. We let n_j be the length of the bucket containing e at the end of the jth partitioning step. Since we choose a random sample of $\sqrt{n_j}$ elements during the execution of the $(j + 1)$st step, we expect n_{j+1} to be approximately $\sqrt{n_j}$. We start by showing the following bound on the relative sizes of the jth and the $(j + 1)$st buckets containing e.

Claim2: $Pr\{n_{j+1} \geq 8n_j^{3/4}\} \leq 2e^{-4n_j^{1/4}}$.

Proof of Claim2: Consider the subsets $B_{j1}, B_{j2}, \ldots$ of the bucket B_j, where each B_{ji} contains $4n_j^{3/4}$ elements of B_j such that each element x of B_{ji} satisfies $4(i-1)n_j^{3/4} \leq rank(x{:}B_j) < 4in_j^{3/4}$. Then, $Pr\{n_{j+1} \geq 8n_j^{3/4}\}$ is bounded by the probability that there exists an index i such that none of the sampled keys during the $(j + 1)$st partitioning step are in B_{ji}. For any fixed index i, the probability that any particular sample key is not among the elements of B_{ji} is precisely $(n_j - 4n_j^{3/4})/n_j = 1 - (4/n_j^{1/4})$. Since the keys are sampled independently, the probability that none of the sampled keys are in B_{ji} is given by

$$\left(1 - \frac{4}{n_j^{1/4}}\right)^{\sqrt{n_j}} \leq e^{-4n_j^{1/4}},$$

using Eq. 9.7. Let e be in B_{ji}. We conclude that the probability that none of the sampled keys are in $B_{j,i-1}$ or in $B_{j,i+1}$ is at most $2e^{-4n_j^{1/4}}$, and therefore

$$Pr\{n_{j+1} \geq 8n_j^{3/4}\} \leq 2e^{-4n_j^{1/4}}.$$

This result concludes the proof of Claim2.

Given the element e, the $(j + 1)$st partitioning step will be called *successful* if $n_{j+1} < 8n_j^{3/4}$. Hence, Claim2 could be interpreted as stating that the probability of an unsuccessful $(j + 1)$st partitioning step is at most

$2e^{-4n_j^{1/4}}$. However, given our assumption that $n_j \geq 16(\log n)^4$, the probability of an unsuccessful partitioning step is at most $2e^{-4(16(\log n)^4)^{1/4}} \leq n^{-7}$.

Claim3: Given an arbitrary input element e, the probability that every partitioning step involving e is successful is at least $1 - O(n^{-6})$.

Proof of Claim3: In Claim2, we established the fact that the probability of an unsuccessful partitioning step is at most n^{-7}. Since the probability of the union of a set of events is bounded by the sum of their probabilities (Boole's inequality), we conclude that the probability there exists one or more unsuccessful partitioning steps is at most n^{-6}. Therefore, Claim3 follows.

Using Claim3, we know that, with high probability, every partitioning step relative to an element e is successful. An argument similar to the one used in the proof of Claim3 shows that every element goes through only successful partitioning steps with high probability. Therefore, with high probability, the total running time is given by $O(\sum_j \log n^{(3/4)^j}) = O(\sum_j (3/4)^j \log n) = O(\log n)$, and, with high probability, the total number of operations is given by $O(\sum_j n \log n^{(3/4)^j}) = O(n \log n)$. The proof of the theorem follows. □

PRAM Model: Algorithm 9.6 requires the CREW PRAM model to run within the stated bounds. In particular, step 2 may require several processors to access the same entry of A, and the process of identifying the bucket number of each element A requires simultaneous access to the sorted sample S. □

Remark 9.2: We can replace the integer sorting required to implement step 4 of Algorithm 9.6 by allocating a separate block of memory for each bucket followed by assigning a random location in the ith block for each element whose bucket number is i. Unfortunately, such a scheme seems to require the CRCW PRAM model. Related details are left to Exercise 9.30. □

9.7 Maximum Matching

A **matching** M in a graph $G = (V, E)$ is a subset of edges such that no two edges from M are incident on the same vertex. Hence, a matching can be viewed as the process of pairing up vertices such that each pair is connected by an edge and no two pairs intersect. The **maximum-matching problem** is to determine a matching M of G such that G contains no other matching M', where $|M'| > |M|$. Of particular interest is the case where G has a **perfect matching**—that is, a matching M such that every vertex of G is incident on an

edge of M. It follows that a perfect matching consists of $|V|/2$ edges and is always maximum.

EXAMPLE 9.18:

Consider the two graphs shown in Fig. 9.4. The graph illustrated in Fig. 9.4(a) has the indicated maximum matching; hence, it contains no perfect matchings. In contrast, the graph shown in Fig. 9.4(b) has a perfect matching, as indicated. □

The computational complexity of the maximum-matching problem has received a considerable amount of attention due to the problem's rich area of applications. In fact, many resource-allocation problems can be formulated as a maximum-matching problem on a graph G that is *bipartite*. As a result, efficient sequential matching algorithms are well known, especially for the case when the graph is bipartite. A basic technique that allows the derivation of polynomial-time sequential algorithms is the **augmenting-path technique,** which will be introduced in Section 10.5 in the context of network flows.

In this section, we shall develop a simple randomized parallel algorithm to compute a maximum matching in an arbitrary graph. The algorithm relies on a generalization of an algebraic characterization of graphs containing a perfect matching, which, in particular, reduces the problem of testing whether a perfect matching exists to the problem of testing whether the determinant of a matrix is identically zero. The randomized parallel algorithms described in this section run in $O(\log^2|V|)$ time, using a polynomial number of operations. Unfortunately, our algorithms use an excessive number of operations compared to those used by the best known sequential algorithms (see bibliographic notes). However, we should mention that there is no known deterministic parallel algorithm running in polylogarithmic time

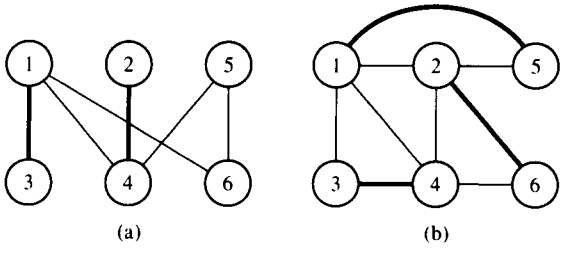


FIGURE 9.4
Two graphs for Example 9.18 (a) A graph with a maximum matching, as indicated by the dark edges. (b) A graph that has the perfect matching consisting of the dark edges.

and using a polynomial number of operations solving the maximum-matching problem. In addition, the algebraic characterization related to graph matchings and the randomization techniques used by the algorithms described in this section are interesting in their own right.

We begin by providing an algebraic characterization in terms of matrix determinants for the existence of a perfect matching in a graph.

9.7.1 PERFECT MATCHINGS AND DETERMINANTS

The two notions of graph matching and matrix determinant seem to be unrelated. Tutte discovered a relationship between the determinant of a matrix associated with a graph G and the existence of a perfect matching in G. The key to this relationship is the following definition of the Tutte matrix associated with G.

Given a graph $G = (V, E)$, let A be the $n \times n$ adjacency matrix of G, where $|V| = n$. Hence, the entry $A(i,j)$ is equal to 1 if and only if there exists an edge in E connecting vertices i and j. Let $\{x_{ij} \mid (i,j) \in E \text{ and } i < j\}$ be a set of indeterminates. The **Tutte matrix** T is the $n \times n$ matrix derived from the adjacency matrix A by substituting x_{ij} for each entry $A(i,j) = 1$ such that $i < j$, and $-x_{ji}$ for each entry $A(i,j) = 1$ such that $i > j$. That is, each nonzero (i, j)-entry above the diagonal is replaced by x_{ij}, and each nonzero (i, j)-entry below the diagonal is replaced by $-x_{ji}$. Notice that, whereas the adjacency matrix A is symmetric, the Tutte matrix T is *skew-symmetric*.

EXAMPLE 9.19:

The Tutte matrices corresponding to the graphs shown in Fig. 9.4(a) and (b) are the following:

$$T_1 = \begin{bmatrix} 0 & 0 & x_{13} & x_{14} & 0 & x_{16} \\ 0 & 0 & 0 & x_{24} & 0 & 0 \\ -x_{13} & 0 & 0 & 0 & 0 & 0 \\ -x_{14} & -x_{24} & 0 & 0 & x_{45} & 0 \\ 0 & 0 & 0 & -x_{45} & 0 & 0 \\ -x_{16} & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$T_2 = \begin{bmatrix} 0 & x_{12} & x_{13} & x_{14} & x_{15} & 0 \\ -x_{12} & 0 & 0 & x_{24} & x_{25} & x_{26} \\ -x_{13} & 0 & 0 & x_{34} & 0 & 0 \\ -x_{14} & -x_{24} & -x_{34} & 0 & 0 & x_{46} \\ -x_{15} & -x_{25} & 0 & 0 & 0 & 0 \\ 0 & -x_{26} & 0 & -x_{46} & 0 & 0 \end{bmatrix}.$$

□

Before establishing a precise relationship between the Tutte matrix T and the existence of a perfect matching in $G = (V, E)$, let us digress for a moment and examine the determinant of T more closely.

The Determinant of the Tutte Matrix. Let S_n denote the group of all permutations on n elements. Then, the determinant of the matrix T is a multivariate polynomial of degree n that is defined by the following sum

$$\det(T) = \sum_{\pi \in S_n} sgn(\pi) t_{1\pi(1)} t_{2\pi(2)} \cdots t_{n\pi(n)},$$

where $sgn(\pi) = \pm 1$, depending on whether the permutation π is even or odd. A permutation is **even** (**odd**) if it can be factored as a product of an even (odd) number of transpositions, where a transposition is a cycle of length 2 (see the next paragraph). We shall denote the product $t_{1\pi(1)} t_{2\pi(2)} \cdots t_{n\pi(n)}$ by $val(\pi)$.

Given an index i such that $1 \leq i \leq n$, the *orbit* of i under the permutation π consists of the elements $i, \pi(i), \pi(\pi(i)) = \pi^2(i), \ldots$. Clearly, there exists a positive integer $l \leq n$ such that $\pi^l(i) = i$, and hence the orbit of i is precisely the set of elements $i, \pi(i), \pi^2(i), \ldots, \pi^{l-1}(i)$. The **cycle** of i is the ordered set $(i, \pi(i), \pi^2(i), \ldots, \pi^{l-1}(i))$. It is easy to see that any permutation can be expressed uniquely as a product of disjoint cycles.

Let π be a permutation such that $val(\pi) \neq 0$. Hence, $t_{i\pi(i)} \neq 0$ for every i; that is, $(i, \pi(i)) \in E$ for every i, where $1 \leq i \leq n$. The subgraph determined by the edges $(i, \pi(i))$ is called the **trail** of π. The cycle of i given by $(i, \pi(i), \pi^2(i), \ldots, \pi^{l-1}(i))$ of π induces a cycle in the trail of π of length l. Notice that, when $l = 2$, the cycle of the trail consists of a single edge traversed twice. Given that any permutation can be expressed uniquely as a product of disjoint cycles, the trail of a permutation π consists of a set of disjoint cycles, each with the same length as the corresponding cycle in the permutation π.

EXAMPLE 9.20:

The Tutte matrix of the graph shown in Fig. 9.5 is given by

$$T = \begin{bmatrix} 0 & 0 & x_{13} & x_{14} \\ 0 & 0 & x_{23} & x_{24} \\ -x_{13} & -x_{23} & 0 & x_{34} \\ -x_{14} & -x_{24} & -x_{34} & 0 \end{bmatrix}.$$

The determinant of T is given by

$$\det(T) = x_{13}^2 x_{24}^2 + x_{14}^2 x_{23}^2 - 2x_{13}x_{24}x_{14}x_{23}.$$

The term $x_{13}^2 x_{24}^2$ corresponds to the permutation $\pi(1) = 3, \pi(2) = 4, \pi(3) = 1$, and $\pi(4) = 2$, which can be expressed as the product (13)(24). The trail of π is thus the perfect matching $\{(1, 3), (2, 4)\}$.

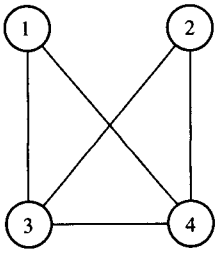


FIGURE 9.5
The Tutte matrix T. The determinant of the Tutte matrix T of the graph shown is given by $\det(T) = x_{13}^2 x_{24}^2 + x_{14}^2 x_{23}^2 - 2x_{13}x_{24}x_{14}x_{23}$. The first two terms correspond to the two perfect matchings $M_1 = \{(1, 3), (2, 4)\}$ and $M_2 = \{(1, 4), (2, 3)\}$, and the last term corresponds to the cycle of length 4 given by $\{(1, 3), (3, 2), (2, 4), (4, 1)\}$.

On the other hand, the term $-x_{13}x_{24}x_{14}x_{23}$ (which occurs twice in the sum defining $\det(T)$) corresponds to the permutation $\sigma(1) = 3, \sigma(2) = 4, \sigma(3) = 2$, and $\sigma(4) = 1$, which consists of the single even cycle (1324). The trail of σ is the subgraph whose edges are $\{(1, 3), (3, 2), (2, 4), (4, 1)\}$, which is a cycle of length 4. □

Tutte's Theorem. The following lemma will be used to establish **Tutte's theorem** relating the determinant of the Tutte matrix T to the existence of a perfect matching in G.

Lemma 9.8: *Let $G = (V, E)$ be a graph on n vertices whose Tutte matrix is T. Then, G has a perfect matching if and only if there exists a permutation π such that $val(\pi) \neq 0$ and π does not have an odd cycle.*

Proof: Suppose π is a permutation in S_n such that $val(\pi) \neq 0$ and π does not contain an odd cycle. The discussion preceding the lemma implies that the trail of π consists of a number of disjoint cycles, each of which is of even length. Hence, we can form a perfect matching M by including in M every other edge of each cycle of length greater than 2 and the edge involved in each cycle of length 2.

We now show the converse. Suppose that G has a perfect matching M. Define the *permutation π_M corresponding to M* as follows: $\pi_M(i) = j$ if and only if $(i,j) \in M$. Hence, π_M consists of $n/2$ cycles, each of length 2. Moreover, $val(\pi_M) = (-1)^{n/2} \prod_{\substack{(i,j)\in M \\ i<j}} x_{ij}^2 \neq 0$. Therefore, the lemma follows. □

Tutte's theorem, which is the main result of this section, is established next.

Theorem 9.13 (Tutte): *Let $G = (V, E)$ be a graph and let T be its Tutte matrix. Then,* $\det(T) \neq 0$ *if and only if G has a perfect matching.*

Proof: Let G have the perfect matching M, and let π_M be the corresponding permutation, as defined in the proof of Lemma 9.8. Then, $val(\pi) = (-1)^{n/2} \prod_{\substack{(i,j) \in M \\ i<j}} x_{ij}^2$. We set $x_{ij} = 1$ whenever $(i,j) \in M$, and $x_{ij} = 0$ otherwise. Then, $val(\sigma) = 0$ for any permutation $\sigma \neq \pi_M$, and hence the value of the determinant is equal to $sgn(\pi_M) val(\pi_M) = 1$, which implies that $\det(T)$ is not identically zero.

Suppose now that $\det(T) \neq 0$. We show that G must have a perfect matching.

Consider any permutation $\sigma \in S_n$ such that $val(\sigma) \neq 0$ and σ contains an odd cycle. If we traverse the odd cycle in the reverse order and leave all the other cycles intact, we obtain another permutation σ' such that $val(\sigma') = -val(\sigma)$ and $sgn(\sigma') = sgn(\sigma)$. For example, the odd cycle $c = (123)$ can be replaced by $c' = (321)$, and hence $val(c) = t_{12}t_{23}t_{31}$ and $val(c') = t_{32}t_{21}t_{13} = -val(c)$, because T is skew-symmetric. It follows that the permutations that contain an odd cycle cancel out each other. Therefore, the fact that $\det(T)$ is not identically zero implies the existence of a permutation π such that $val(\pi) \neq 0$ and π contains no odd cycles. By Lemma 9.8, G has a perfect matching. □

9.7.2 RANDOMIZED ALGORITHM FOR TESTING THE EXISTENCE OF A PERFECT MATCHING

Tutte's theorem (Theorem 9.13) reduces the problem of testing whether a graph $G = (V, E)$ has a perfect matching to that of testing whether $\det(T)$ is identically zero, where T is the Tutte matrix. We developed, in Section 9.5, a randomized scheme for testing polynomial identities. This scheme will be adapted to solve our current problem as follows.

Let I be the set of integers $\{-n, -n+1, \ldots, n\}$. For each variable x_{ij}, assign a value from I uniformly and independently. We compute the determinant of the integer matrix obtained from the Tutte matrix by substituting the values of the indeterminates. If the value of the determinant is nonzero, we declare that G has a perfect matching; otherwise, we declare that G does not have a perfect matching. The probability of error is $< 1/2$, and hence we can make this error arbitrarily small by repeating the same process several times.

The complexity bounds of our randomized algorithm are dominated by those for computing the determinant of an integer matrix. We state without proof the following theorem, which will also be used later to determine a

perfect matching, whenever one exists. The **adjoint** $adj(A)$ of an $n \times n$ matrix A is an $n \times n$ matrix whose (i,j) entry is equal to $(-1)^{i+j} \det(A_{ji})$, where A_{ji} is the submatrix obtained from A by deleting the jth row and the ith column.

Theorem 9.14: *Given an $n \times n$ integer matrix A such that each entry of A is represented by a k-bit integer, then $\det(A)$ and $adj(A)$ can be computed by a randomized parallel algorithm in $O(\log^2 n)$ time, using a total of $O(nM(n)k)$ operations, where $M(n)$ is the number of arithmetic operations used by the best known sequential algorithm for multiplying two $n \times n$ matrices.* □

Remark 9.3: The total number of operations referred to in Theorem 9.14 is estimated in the Boolean model, since the sizes of the integers encountered while computing the determinant of a matrix or its inverse can become quite large. We also have ignored logarithmic factors in this count, since they can be absorbed by the $M(n)$ bound. Currently, the best known bound of $M(n)$ is $M(n) = O(n^{2.376})$. □

In particular, we obtain the following corollary.

Corollary 9.5: *Given a graph $G = (V, E)$, we can test whether G has a perfect matching by using a randomized parallel algorithm running in $O(\log^2 n)$ time and using $O(n^{3.5})$ operations.* □

9.7.3 DETERMINATION OF A MAXIMUM MATCHING

The randomized algorithm referred to in Corollary 9.5 does not find a perfect matching in G if one exists. It merely indicates the existence or nonexistence of a perfect matching. We shall expand some of the facts established in Section 9.7.1 to derive a randomized algorithm for identifying the edges in a perfect matching. We shall later show how to use this algorithm to determine maximum matchings in general graphs.

In the remainder of this section, we assume that our input $G = (V, E)$ has a perfect matching. Our goal is to develop a randomized scheme for identifying such a matching. The main stumbling block seems to be that G may have many perfect matchings, which makes it difficult to coordinate the concurrent processing of G to converge toward a single perfect matching. We will avoid this difficulty by assigning random integer weights to the edges of G, which will guarantee the uniqueness of a minimum-weight perfect matching with probability at least 1/2. This technique is introduced next in a slightly more general context.

The Isolating Lemma. A **set system** consists of a pair (S, F), where S is a finite set of elements, $S = \{e_1, e_2, \ldots, e_n\}$, and F is a collection of subsets of S; that is, $F = \{S_1, S_2, \ldots, S_t\}$, where $S_j \subseteq S$ for $1 \leq j \leq t$.

We assign an integer weight w_i to each element $e_i \in S$. The weight of a subset $S_j \in F$ is defined by $w(S_j) = \sum_{e_i \in S_j} w_i$. Our goal is to assign the weights such that we make the minimum-weight subset in F unique. A randomized scheme is described in the next lemma.

Lemma 9.9: *Let (S, F) be a set system such that each element of S is assigned a random integer weight from the range $[1, 2, \ldots, 2n]$, where n is the number of elements in S. Then, the probability that the minimum-weight subset in F is unique is at least 1/2.*

Proof: We show that, with probability at least 1/2, each element of S is either (1) in every minimum-weight subset in F, or (2) in no such subset. The property that each element is either in every minimum-weight subset or in no such subset would imply that the minimum-weight subset is unique.

We partition the subsets in F into two groups G_1 and G_2 such that G_1 consists of all the subsets $S_j \in F$ containing e_i, and G_2 consists of the remaining subsets in F. Suppose that certain weights have been assigned to all the elements of S except e_i. Let δ_1 and δ_2 be the minimum weights of the subsets in G_1 and G_2, respectively, where the weight of a subset in G_1 is the sum of its elements' weight with the exception of e_i. If $\delta_1 \geq \delta_2$, then the element e_i will not be in any minimum-weight subset, regardless of the weight assigned to e_i.

Consider the more interesting case where $\delta_1 < \delta_2$. Then, e_i belongs to a minimum-weight subset if and only if $w_i + \delta_1 \leq \delta_2$; that is, $w_i \leq \delta_2 - \delta_1$. In fact, e_i belongs to every minimum-weight subset whenever $w_i < \delta_2 - \delta_1$, and e_i does not belong to any minimum-weight subset whenever $w_i > \delta_2 - \delta_1$. The only problem occurs when $w_i = \delta_2 - \delta_1$, since there exists a minimum-weight subset that contains e_i and another that does not contain e_i. We call e_i **ambiguous** in the latter case.

Since w_i is chosen uniformly and independently from the range $[1, 2, \ldots, 2n]$, the value of w_i is independent of $\delta_2 - \delta_1$. Thus, as long as w_i misses this particular value, e_i will be unambiguous, which implies that

$$Pr\{e_i \text{ is ambiguous}\} \leq \frac{1}{2n}.$$

It follows that the probability that there exist one or more ambiguous elements in S is at most $n \times (1/2n) = 1/2$. Hence, with probability at least 1/2, each element is either in every minimum-weight subset in F or in no such subset; hence, the minimum weight subset is unique. □

Determination of a Perfect Matching. Let $G = (V, E)$ be a graph that contains a perfect matching. The pair $(E, \mathcal{M})$, where $\mathcal{M}$ is the set of all perfect matchings in G, forms a set system. We assign random integer weights to the edges of G, chosen uniformly and independently from the range $[1, 2, \ldots, 2m]$, where $m = |E|$. By the isolating lemma (Lemma 9.9), there exists a unique minimum-weight perfect matching in G. We shall develop a parallel algorithm to identify this particular perfect matching.

We start by providing a simple method for computing the weight of the unique minimum-weight perfect matching. Let T be the Tutte matrix associated with the graph G. We substitute the integer $2^{w_{ij}}$ for the indeterminate x_{ij}, where w_{ij} is the weight assigned to the edge $(i,j) \in E$. We denote the resulting matrix by $\mathcal{T}$.

Lemma 9.10: *Let $G = (V, E)$ be a weighted graph such that G has a unique minimum-weight perfect matching—say, of weight W. Then,* $\det(\mathcal{T}) \neq 0$, *and the highest power of 2 that divides* $\det(\mathcal{T})$ is 2^{2W}, *where $\mathcal{T}$ is the matrix we obtain from the Tutte matrix by substituting $2^{w_{ij}}$ for the indeterminate x_{ij}, and where w_{ij} is the weight of edge (i, j).*

Proof: Let S'_n be the set of permutations on n elements such that no permutation of S'_n contains an odd cycle. As we remarked in the proof of Lemma 9.8, the permutations containing odd cycles do not contribute to the value of $\det(T)$, where T is the Tutte matrix of the graph G.

Let M be the unique minimum-weight perfect matching, and let π_M be the corresponding permutation. Using the values of the entries of $\mathcal{T}$, we obtain that $val(\pi_M) = (-1)^{n/2} 2^{2W}$. Consider now any other permutation $\sigma \in S'_n$. If each cycle of σ is of length 2, then $val(\sigma) = (-1)^{n/2}\, 2^{2w(M')}$, where M' is the perfect matching associated with σ, and where $w(M')$ is its weight. Since M is the unique minimum-weight perfect matching, we obtain that $w(M') > W$; hence, $val(\sigma)$ is divisible by 2^{2W}.

Suppose that σ has cycles of even length greater than 2. Then, we can deduce two perfect matchings M_1 and M_2 from σ by choosing complementary sets of edges from each cycle of length greater than 2. Hence, $|val(\sigma)| = 2^{w(M_1)+w(M_2)}$, where each of $w(M_1)$ and $w(M_2)$ is larger than W. It follows that $val(\sigma)$ is divisible by 2^{2W} as well.

It follows that each term of the sum defining $\det(\mathcal{T})$ is divisible by 2^{2W}, and that there is one term—namely, $val(\pi_M)$—that is precisely equal to 2^{2W}. Therefore, 2^{2W} is the highest power of 2 that divides $\det(\mathcal{T})$. □

Lemma 9.10 enables us to determine the weight of the minimum-weight perfect matching once the determinant of the integer matrix $\mathcal{T}$ has been

computed. We now characterize all the edges that belong to the unique minimum-weight perfect matching.

In what follows, we use the notation $\mathcal{T}_{ij}$ to indicate the submatrix of $\mathcal{T}$ that we obtain by removing the ith row and the jth column from $\mathcal{T}$.

Theorem 9.15: *Let $G = (V, E)$ be a weighted graph that has a unique minimum-weight perfect matching of weight W. Then, the edge (i, j) belongs to M if and only if $2^{w_{ij}} \det(\mathcal{T}_{ij})/2^{2W}$ is odd.*

Proof: Consider the set of all permutations $\pi \in S_n$ such that $\pi(i) = j$. Then, the sum $\sum_{\pi(i)=j}^{\pi \in S_n} sgn(\pi)val(\pi)$ generates precisely the terms defining $\det(\mathcal{T}_{ij})$ such that each such term is multiplied by $\mathcal{T}(i, j) = 2^{w_{ij}}$. Therefore, the following equation holds:

$$2^{w_{ij}} \det(\mathcal{T}_{ij}) = \sum_{\substack{\pi \in S_n \\ \pi(i)=j}} sgn(\pi)val(\pi). \tag{9.8}$$

Consider any permutation π containing an odd cycle such that $\pi(i) = j$. Since n is even, π must contain at least two odd cycles. By reversing the order of an odd cycle, we can derive another permutation π' such that $\pi'(i) = j$, $val(\pi') = -val(\pi)$, and $sgn(\pi') = sgn(\pi)$ (as in the proof of Theorem 9.13). Hence, the permutations containing odd cycles cancel out each other in Eq. 9.8. Therefore, we can assume that our sum consists of permutations containing only even cycles.

Suppose that edge (i,j) belongs to the minimum-weight perfect matching M. Then, the corresponding permutation π_M satisfies $\pi_M(i) = j$, and hence it is included in the sum of Eq. 9.8. The value of any other term that corresponds to a permutation containing no odd cycles is 2^{α}, where $\alpha > 2W$. Therefore, $2^{w_{ij}} \det(\mathcal{T}_{ij}) = 2^{2W} + 2^{\alpha_1} + \ldots + 2^{\alpha_t}$, for some t, where $\alpha_i > 2W$ for each $1 \leq i \leq t$. It follows that $2^{w_{ij}} \det(\mathcal{T}_{ij})/2^{2W}$ is odd.

Conversely, suppose that $2^{w_{ij}} \det(\mathcal{T}_{ij})/2^{2W}$ is odd, and yet (i,j) is not in the minimum-weight perfect matching. For any $\pi \in S_n$ such that $\pi(i) = j$ and π does not contain an odd cycle, $|val(\pi)| = 2^{2w(M')}$ or $2^{w(M_1)+w(M_2)}$, for some perfect matchings M', M_1, and M_2 (as in the proof of Lemma 9.10). Therefore, the sum defining Eq. 9.8 can be expressed as $\sum_i 2^{\alpha_i}$, where $\alpha_i > 2W$. This fact implies that $2^{w_{ij}} \det(\mathcal{T}_{ij})/2^{2W}$ is even, which contradicts our hypothesis. Therefore, (i, j) must be in the minimum-weight perfect matching. □

We are ready to describe our randomized algorithm for determining a perfect matching.

ALGORITHM 9.7
(Perfect Matching)
Input: *A graph $G = (V, E)$, such that G has a perfect matching.*

Output: *A set M of edges such that M is a perfect matching with probability* $\geq 1/2$.
begin
1. Assign random integer weights w_{ij} to the edges $(i, j) \in E$, such that each weight is chosen uniformly and independently from $[1, 2, \ldots, 2m]$ and $m = |E|$.
2. Let $\mathcal{T}$ be the matrix derived from the Tutte matrix of G by substituting $2^{w_{ij}}$ for each indeterminate x_{ij}.
3. Compute $\det(\mathcal{T})$, *and deduce the value W of the weight of the minimum-weight perfect matching.*
4. Compute the adjoint matrix $adj(\mathcal{T})$, where the absolute value of its (i, j)th entry is the determinant of $\mathcal{T}_{ji}$.
5. **for** *each* $(i, j) \in E$ **pardo**
if $(2^{w_{ij}} \det(\mathcal{T}_{ij})/2^{2W}$ *is odd*) **then** *include* (i, j) *in* M.
end

The main computational steps of Algorithm 9.7 are those involving the computations of $\det(\mathcal{T})$ and $adj(\mathcal{T})$. Theorem 9.14 provides the complexity bounds for such computations. Therefore, we have the following theorem.

Theorem 9.16: *Algorithm 9.7 determines a set M of edges such that the probability that M is a perfect matching of the input graph $G = (V, E)$ is at least* $1/2$. *This algorithm can be implemented to run in* $O(\log^2 n)$ *time, using* $O(n^{3.5}m)$ *operations, where* $|V| = n$ and $|E| = m$. □

Maximum Matching. Up to this point, we have focused on determining a perfect matching whenever our input graph has one. We now show how to reduce the problem of determining a maximum matching in an arbitrary graph $G = (V, E)$ to the problem of determining a perfect matching in another graph derived from G.

Let α be the size of the maximum matching M of G. Then, G has 2α vertices that are matched in M, and $n - 2\alpha$ vertices that are unmatched. We add $n - 2\alpha$ new vertices such that each new vertex is connected to each $v \in V$. Let us denote the new graph by $G' = (V', E')$. Clearly, G' has a perfect matching containing M. Moreover, every perfect matching of G' contains a matching of G of size α. Hence, if we know α, we can use Algorithm 9.7 to compute a maximum matching in G.

In general, we do not know the size of the maximum matching. We use the following scheme. Let k be any integer such that $1 \leq k \leq n - 2$ and $k + n$ is even. Then, we add k new vertices to G, and connect each of the new vertices to all the vertices in the original graph G. We denote the resulting graph by G_k. It is easy to check that the resulting graph G_k has a perfect matching if and only if $k \geq n - 2\alpha$, where α is the size of a maximum matching

in G. Therefore, we can solve the problem of determining the maximum matching of G by performing a binary search on the minimum value of k for which the graph G_k has a perfect matching. We then compute a perfect matching to deduce a maximum matching in G.

Corollary 9.6: *A maximum matching of a graph G with n vertices and m edges can be determined by a randomized parallel algorithm in $O(\log^3 n)$ time, using a total of $O(n^{3.5}m)$ operations.* □

Another scheme for reducing the maximum-matching problem to the problem of computing a perfect matching is left to Exercise 9.36. We note here only that this scheme leads to an $O(\log^2 n)$ time parallel randomized algorithm for computing the maximum matching, compared to the $O(\log^3 n)$ time bound obtained by the algorithm described here.

TABLE 9.1
ALGORITHMS DISCUSSED IN THIS CHAPTER.

Algorithm		Section	$T(n)$	$W(n)$	PRAM Model
9.1	Randomized Symmetry Breaking (Directed Cycles)	9.2.1	$O(1)$	$O(n)$	EREW
9.2	Fractional Independent Set	9.2.2	$O(1)$	$O(n)$	CREW
9.3	Subdivision Hierarchy	9.3.2	$O(\log n)$	$O(n)$	CREW
9.4	Monte Carlo String Matching	9.4.2	$O(\log n)$	$O(n)$	EREW
	Two-dimensional Array Matching	9.4.3	$O(\log n)$	$O(n^2)$	EREW
	Verification of Matrix Products	9.5.2	$O(\log n)$	$O(n^2)$	CREW
9.5	Randomized Quicksort	9.6.3	$O(\log^2 n)$	$O(n \log n)$	EREW
9.6	$\sqrt{n}$ Sample Sort	9.6.4	$O(\log n)$	$O(n \log n)$	CREW
	Testing the Existence of a Perfect Matching	9.7.2	$O(\log^2 n)$	$O(n^{3.5})$	CREW
9.7	Perfect Matching	9.7.4	$O(\log^2 n)$	$O(n^{3.5}m)$	CREW
	Maximum Matching	9.7.4	$O(\log^3 n)$	$O(n^{3.5}m)$	CREW

m: number of edges

9.8 Summary

We have seen a sample of simple parallel algorithms that use randomization. We have touched on several general techniques that can be classified as **symmetry breaking, algebraic substitution, random sampling,** and **forcing uniqueness.** These algorithms offer attractive alternatives to the corresponding deterministic algorithms. Many additional results are contained in the references given at the end of this chapter.

Table 9.1 gives a summary of the complexity bounds of the algorithms developed in this chapter.

Exercises

9.1. Let S be a sample space with a probability measure. Use the axioms of a probability measure to show the following:

a. Given any two events $A \subseteq B$, then $Pr\{A\} \leq Pr\{B\}$.

b. Given any two events A and B, then $Pr\{A \cup B\} = Pr\{A\} + Pr\{B\} - Pr\{A \cap B\}$.

c. Given any collection of pairwise disjoint events $\{A_i\}_{i=1}^{k}$, then $Pr\{\cup_i A_i\} = \sum_i Pr\{A_i\}$.

d. Given an arbitrary collection of events $\{A_i\}$, then $Pr\{\cup_i A_i\} \leq \sum_i Pr\{A_i\}$. This relation is called **Boole's inequality.**

9.2. Let S be a sample space that is partitioned into disjoint sets $\{A_i\}$, and let B be any event such that $Pr\{B\} > 0$.

a. Show that $Pr\{B\} = \sum_i Pr\{A_i\}Pr\{B \mid A_i\}$.

b. Show that

$$Pr\{A_i \mid B\} = \frac{Pr\{A_i\}Pr\{B|A_i\}}{\sum_i Pr\{A_i\}Pr\{B|A_i\}}.$$

This equation is a formulation of **Bayes' theorem.**

9.3. Consider the experiment of randomly distributing m balls into n bins. Let E_i be the event that bin i remains empty; clearly, $Pr\{E_i\} = (1 - \frac{1}{n})^m$. Can we deduce that the probability of at least one of the bins remains empty is equal to $\sum_{i=1}^{n} Pr\{E_i\} = n\left(1 - \frac{1}{n}\right)^m$? Justify your answer.

9.4. Let X be a random variable assuming only nonnegative values. Show the following **Markov inequality:**

$$Pr\{X \geq k\mu_X\} \leq \frac{1}{k}.$$

9.5. Prove Theorem 9.1.

9.6. Given a sequence of independent Bernoulli trials, each with a success probability p, show that the expected number of trials needed to obtain a success is $1/p$.

9.7. Let X be the number of successes in n Bernoulli trials, each with a success probability p. Show that the variance of X is given by $\sigma_X^2 = np(1 - p)$.

9.8. Consider the experiment of randomly distributing m balls into n bins. What is the probability that a bin contains more than a single ball? Show that the corresponding probability can be bounded by $e^{-m(m-1)/2n}$ by using the fact that $1 + x \leq e^x$.

9.9. Consider the experiment of randomly distributing m *identical* balls into n bins, where $m > n$. What is the probability that every bin contains at least one ball?

9.10. Suppose that m labeled balls are randomly placed into $2m$ bins. Show that, with high probability, a constant fraction of the bins will contain a single ball.

9.11. Let L be a linked list of n nodes specified by an array S such that $S(i)$ is a pointer to the successor of i. Recall from Section 3.1 that the list-ranking problem for L is to determine the distance $R(i)$ of each node i from the end of the list. Using the randomized symmetry-breaking technique, develop a list-ranking algorithm that runs in $O(\log n \log \log n)$ time with high probability. What is the total number of operations used?

9.12. Rewrite Algorithm 9.2 so that it runs on the EREW PRAM within the same complexity bounds stated in Theorem 9.3.

9.13. Let $G = (V, E)$ be a graph such that each vertex of G is of degree at most δ. Develop a randomized parallel algorithm to color the vertices of G using at most $\delta + 1$ colors. With high probability, your algorithm should run in $O(\log n)$ time, using a linear number of operations.

9.14. Prove Lemma 9.3.

9.15. Prove Theorem 9.6 and Corollary 9.2.

9.16. Develop a Las Vegas string-matching algorithm that runs in $O(\log n)$ time and uses $O(n)$ operations with high probability, assuming that the pattern is nonperiodic. Recall that a string $P(1{:}m)$ is called *nonperiodic* whenever its period is greater than $m/2$.

9.17. Suppose that a constant number $c > 1$ of random primes are chosen at step 2 of Algorithm 9.4. What is the corresponding error probability?

9.18. Generalize the randomized algorithm for the two-dimensional array matching (Section 9.4.3) to d-dimensional array matching, where d is constant.

9.19. Consider the two-dimensional pattern-matching problem, where the pattern is not necessarily rectangular (but the text is assumed to be a square of size $n \times n$). Develop parallel randomized algorithms to handle these cases: (a) the pattern P is diamond-shaped, and (b) the pattern P is L-shaped. What are the corresponding complexity bounds? What is the error probability in each case?

9.20. Develop a deterministic parallel algorithm to determine whether the set $\{A^i\}_{i=0}^{n-1}$ is linearly dependent, where A is an $n \times n$ matrix. Compare with the randomized algorithm suggested in Example 9.15.

9.21. Let $p(x), q(x)$, and $r(x)$ be three polynomials such that $\deg(p) = \deg(q) = n$ and $\deg(r) = 2n$. Develop a randomized algorithm to test whether $p(x)q(x) = r(x)$. Your algorithm should run in $O(\log n)$ time, using a total of $O(n)$ operations.

9.22. Let A be an $n \times n$ matrix, and let $\phi_A(\lambda)$ be its characteristic polynomial (Section 8.8). The *minimum polynomial* of A is the minimum-degree monic polynomial $p(\lambda)$ such that $p(A) = 0$. The **discriminant** of A is the determinant of the resultant matrix corresponding to $\phi_A(\lambda)$ and $\phi'_A(\lambda)$ (see Section 8.9 for the definition of the resultant matrix).

a. Show that, if the discriminant of A is nonzero, then $\phi_A(\lambda) = p_A(\lambda)$.

b. Assume that all principal minors of A are nonsingular. Develop a randomized algorithm to determine a diagonal matrix D such that $\phi_{AD}(\lambda) = p_{AD}(\lambda)$. Your algorithm should run in $O(\log^2 n)$ time.

9.23. Let $\{a_i\}_{i=0}^{\infty}$ be an infinite sequence of elements from a field F. We say that this sequence is **finitely generated** if there exist $c_0, c_1, \ldots, c_n$ in F, not all of them zeros, such that

$$c_0 a_j + \cdots + c_n a_{j+n} = 0$$

for all $j \geq 0$. The polynomial $c_0 + c_1\lambda + \ldots + c_n\lambda^n$ is called a **generating polynomial** for $\{a_i\}$. The smallest-degree monic generating polynomial is called the **minimum polynomial** of $\{a_i\}$.

Suppose that $\{a_i\}$ is linearly generated, and let m be the degree of its minimum polynomial. For each $s \geq 0$, let T_s be the Toeplitz matrix generated by the vector $(a_0, a_1, \ldots, a_{2s-2})$ (see Section 8.5).

a. Show that $\det(T_m) \neq 0$ and that $\det(T_s) = 0$ for $s > m$.

b. Deduce an efficient parallel algorithm to determine the minimum generating polynomial of the sequence $\{a_i\}$. *Hint*: Use the algorithm referred to in Corollary 8.6.

9.24. Let A be an $n \times n$ matrix over a field F, and let p_A be its minimum polynomial. Given a finite subset S of F, choose two random vectors u and v, and consider the sequence $\{u^T A^i v\}$. Clearly, this sequence is linearly generated (see Exercise 9.23).

a. Show that the probability that the minimum generating polynomial of the sequence $\{u^T A^i v\}$ is equal to the minimum polynomial of A is at least $1 - \frac{2 \deg(p_A)}{|S|}$.

b. Deduce an efficient randomized parallel algorithm to compute the minimum polynomial of A.

c. Deduce an efficient randomized parallel algorithm to determine whether $\det(A) = 0$.

9.25. Let A be a set of n distinct elements drawn from a linearly ordered set. We construct a **random binary search tree** T as follows. We select a random element X of A and make it the root. Remove X from A. Next, we select a random element from the remaining elements, and insert it in the left or right subtree of the root, depending on whether the element is smaller or larger than the root. We continue in this fashion until all the elements of A are exhausted.

Show that, with high probability, the depth of the tree is $O(\log n)$.

9.26. Let the **random-write CRCW PRAM** be a CRCW PRAM such that, whenever several processors attempt to write into a single location, a randomly chosen processor will succeed. Develop a parallel version of the randomized quicksort algorithm (Algorithm 9.5) using binary search trees as follows. The first splitter constitutes the root, the smaller elements are inserted in the left subtree, and the larger elements are inserted in the right subtree. Show how to implement this strategy on a random-write CRCW PRAM such that, with high probability, the algorithm runs in $O(\log n)$, using $O(n \log n)$ operations.

9.27. You are given an array A of length n containing k marked elements. Using the arbitrary CRCW PRAM, show how to select one of the marked elements in $O(1)$ time, with high probability. Each of the marked elements should be equally likely to be selected, and the number of processors used should be n. Assume that k is known.

9.28. Develop a simple algorithm to sort n integers in the range $[1, 2, \ldots, n]$ in $O(\log n)$ time, using $O(n \log n)$ operations. Your algorithm must run on the EREW PRAM. What is the space needed by your algorithm?

9.29. Develop an $O(\log n)$ time randomized algorithm to generate a random permutation of $(1, 2, \ldots, n)$ using a linear number of operations. You can use a CRCW PRAM.

9.30. Let A be an array of size n such that each element is marked with a bucket number in the range $[1, 2, \ldots, m]$, where m divides n. The number of elements belonging to each bucket is exactly n/m. Develop a randomized parallel algorithm to rearrange the elements of A such that the elements in the first bucket appear first, followed by the elements of the second bucket, and so on. Your algorithm should run in $O(\log n)$ time, using a linear number of operations, with high probability. Use the

arbitrary CRCW PRAM with the following scheme. Assign a block B_i of size $2n/m$ to the elements in the ith bucket. Assign a random location in B_i to each element labeled i.

9.31. Consider the problem of computing the maximum of a set X of n elements on a priority CRCW PRAM using the following basic scheme: (1) draw a sample S from X such that each element is included in S with probability $1/(2n^{3/4})$; (2) use a random allocation scheme to store the elements in an array A of length $\sqrt{n}$; (3) compute the maximum m in A; and (4) retain all the elements of X larger than m.

- **a.** Show that the number of retained elements is no more than $n^{3/4} \log n$, with high probability.
- **b.** Show how to implement the basic scheme in $O(1)$ time, with high probability.
- **c.** Derive a randomized algorithm that computes the maximum in constant time, with high probability.

9.32. Assume that in the sample sort algorithm (Algorithm 9.6), step 2 selects each element to be in the sample with probability $1/\sqrt{n}$, independently of the selection of any other element. Show that the resulting algorithm will run in $O(\log n)$ time, using $O(n \log n)$ operations, with high probability.

9.33. Another method (see also Exercise 9.32) of choosing the sample at step 2 of the sample sort algorithm (Algorithm 9.6) is to partition the input into $\sqrt{n}$ subarrays, and to select randomly an element from each subarray. Show that the resulting algorithm will run in $O(\log n)$ time, using $O(n \log n)$ operations, with high probability.

9.34. An **approximate median** of a set A of n elements is an element whose rank is αn, for some constant α such that $0 < \alpha < 1$. Develop an $O(\log n)$ time randomized algorithm to compute an approximate median for A using a linear number of operations. *Hint:* Take a sample S of A of size $\sqrt{n}$, and find the median of S.

9.35. Let X be a set of n elements drawn from a linearly ordered set. Pick s elements of X uniformly and independently—say, $y_1 < y_2 < \cdots < y_s$. Let $rank(y_i{:}X) = r_i$. Show that, for every $\alpha > 0$, we have

$$Pr\{|r_i - i\tfrac{n}{s+1}| > c\alpha\tfrac{n}{\sqrt{s}}\sqrt{\log n}\} < n^{-\alpha},$$

for some constant $c > 0$.

9.36. Develop a randomized algorithm to compute the maximum matching of a graph with n vertices and m edges, such that the running time is $O(\log^2 n)$. *Hint*: Extend G to a complete graph. For each edge e, select a weight $w(e)$ uniformly and independently from the range $[1, 2, \ldots, n^2]$. Add n^3 to the weight of each new edge. Show that the resulting graph has a unique minimum-weight perfect matching with probability at least 1/2.

Bibliographic Notes

Randomized algorithms were formally introduced and popularized by Rabin [37]. Independently, Solovay and Strassen [49] published a randomized algorithm for primality testing. The underlying techniques had been used in Monte Carlo simulations for many years before. In addition, computational complexity had been injected earlier by probabilistic notions [16, 19, 36]. Since then, many randomized algorithms have appeared. Raghavan's lecture notes [39] provide an excellent reference on sequential randomized algorithms. Some of the early work on randomized parallel algorithms was reported in [3, 33, 43, 55]. Concerning background material, the books [7, 14] are good references on probability theory. The bounds on the tail of a binomial distribution given in Eqs. 9.2 and 9.3 follow from the general techniques of Chernoff [4]. For the derivation of these bounds, see [1, 21, 39].

The subdivision-hierarchy approach to solve the point-location problem was presented by Kirkpatrick [27]. The randomized parallel algorithm was discovered independently by Dadoun and Kirkpatrick [11] and by Reif and Sen [44]. Optimal deterministic parallel algorithms appeared in [10, 50]. In general, the randomized sampling technique is a powerful tool for solving computational geometry problems (see, for example [8, 9, 42, 48]). Karp and Rabin introduced the randomized string-matching algorithms based on the fingerprint functions [25]. The proofs of Lemmas 9.4 and 9.5 can be obtained from results appearing in [46]. The technique for verifying polynomial identities was discussed by Schwartz [47]; the specific result concerning the verification of matrix products had been obtained earlier by Frievalds [17].

The quicksort strategy was developed by Hoare [22]. Reischuk [45] generalized this strategy to obtain an optimal $O(\log n)$ time randomized sorting algorithm, similar to the one presented in the text. Our proof of Theorem 9.11 is taken directly from [39]. A considerable amount of work has been done on randomized parallel sorting algorithms; for example, see [2, 5, 6, 20, 30, 41].

Edmonds [13] developed the fundamental techniques to solve efficiently the maximum-matching problem sequentially. The current best known bound on the sequential complexity of the maximum-matching problem is $O(\sqrt{n}m)$ as reported by Micali and Vazirani in [32]. The work by Tutte concerning the relationship between the perfect matching and the determinant of the Tutte matrix appeared in [51]. Lovasz [29] noted that Tutte's theorem can be used to reduce the problem of testing whether a graph has a perfect matching to that of testing whether the determinant of an integer matrix is nonzero. The first randomized parallel algorithm for solving the maximum-matching problem was given by Karp, Upfal, and Widgerson [26]. The efficiency of this algorithm was later improved in [18]. Our presentation of the randomized parallel algorithm for solving the perfect-matching problem (including the isolating lemma) follows [34]. Our reduction of the maximum-matching problem to the perfect-matching problem is taken from [38]. The randomized parallel algorithms for solving the maximum-matching problem can be made into Las Vegas algorithms, as shown in [24]. More detailed coverage on the parallel complexity of the matching probem appeared in [54]. The proof of Theorem 9.14 can be found in [35].

An important application of randomized algorithms has been in the area of *packet routing*. Early work suggesting this approach was reported by Valiant, and Valiant and Brebner in [52, 53].

A solution to Exercise 9.17 can be found in [25]. Frievalds [17] considers the problem stated in Exercise 9.21. Solutions to Exercises 9.23 and 9.24, as well as efficient randomized parallel algorithms for computing the determinant and for solving linear

systems of equations, can be found in [23]. Random binary search trees (Exercise 9.25) have been studied in [12]. A detailed implementation of the algorithm sketched in Exercise 9.26 appeared in [6]. A solution to Exercise 9.28 can be found in [28]. A more general version of the problem discussed in Exercise 9.30 was addressed in [41]. A randomized scheme to handle the general selection problem (Exercise 9.31) was initially proposed by Floyd and Rivest [15]. Randomized parallel algorithms for selection appeared in [31, 45, 47]. The bound stated in Exercise 9.35 was established in [40]. The hint given for Exercise 9.36 is taken from [34].

References

1. Angluin D., and L. G. Valiant. Fast probabilistic algorithms for Hamiltonian circuits and matchings. *JCSS*, 18(2):155-193, 1979.
2. Blelloch, G. E., C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the Connection Machine CM-2. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, Hilton Head, SC, 1991, pp. 3-16.
3. Borodin A., J. von zur Gathen, and J. Hopcroft. Fast parallel matrix and GCD computation. *Information and Control*, 52(3):241-256, 1982.
4. Chernoff, H. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Math. Stat.*, 2:493-509, 1952.
5. Chlebus, B. S. Parallel iterated bucket sort. *Information Processing Letters*, 31(4):181-183, 1989.
6. Chlebus, B. S., and I. Vrto. Parallel quicksort. *Journal of Parallel and Distributed Computing*, 11(4):332-337, 1991.
7. Chung, K. L. *Elementary Probability Theory with Stochastic Processes.* Springer-Verlag, New York, 1985.
8. Clarkson, K. L. New applications of random sampling in computational geometry. *Discrete and Computational Geometry*, 2(2):195-222, 1987.
9. Clarkson, K. L. Applications of random sampling in computational geometry. In *Proceedings Fourth ACM Symposium on Computational Geometry*, Urbana-Champaign, IL, 1988, pp. 1-11.
10. Cole, R., and O. Zajicek. An optimal parallel algorithm for building a data structure for planar point location. *Journal of Parallel and Distributed Computing*, 8(3):280-285, 1990.
11. Dadoun, N., and D. Kirkpatrick. Parallel processing for efficient subdivision search. In *Proceedings Third ACM Symposium on Computational Geometry*, Waterloo, Ontario, Canada, 1987, pp. 205-214.
12. Devroye, L. Branching processes in the analysis of the heights of trees. *Acta Informatica*, 24(3):277-298, 1987.
13. Edmonds, J. Paths, trees and flowers. *Canad. J. Math.*, 17(3):449-467, 1965.
14. Feller, W. *An Introduction to Probability Theory and Its Applications*, Volume I. John Wiley, New York, 1968.
15. Floyd, R., and R. Rivest. Expected time bounds for selection. *Communications of ACM*, 18(3):165-172, 1975.

16. Freivalds, R. Probabilistic machines can use less running time. In B. Gilchrist, editor, *Information Processing*, Amsterdam, North Holland, 1977.

17. Freivalds, R. Fast probabilistic algorithms. In *Proceedings of Mathematical Foundations of Computer Science*, Volume 74. Springer-Verlag, New York, 1979.

18. Galil, Z., and V. Pan. Improved processor bounds for combinatorial problems in RNC. *Combinatorica*, 8(2):189–200, 1988.

19. Gill, J. Computational complexity of probabilistic turing machines. *SIAM J. Computing*, 6(4):675–695, 1977.

20. Hagerup, T. Hybridsort revisited and parallelized. *Information Processing Letters*, 23(1):35–39, 1989.

21. Hagerup, T., and C. Rüb. A guided tour of Chernoff bounds. *Information Processing Letters*, 33(6):305–308, 1990.

22. Hoare, C. A. R. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

23. Kaltofen, E., and V. Pan. Processor efficient parallel solutions of linear systems over an abstract field. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, Hilton Head, SC, 1991, pp. 180–191.

24. Karlof, H. Las Vegas RNC algorithm for maximum matching. *Combinatorica*, 6(4):387–391, 1986.

25. Karp, R. M., and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Develop.*, 31(2):249–260, 1987.

26. Karp, R. M., E. Upfal, and A. Wigderson. Constructing a maximum matching in random NC. *Combinatorica*, 6(1):35–48, 1986.

27. Kirkpatrick, D. G. Optimal search in planar subdivisions. *SIAM J. Computing*, 12(1):28–35, 1983.

28. Kruskal, C., L. Rudolph, and M. Snir. The power of parallel prefix. *IEEE Transactions on Computers*, 34(10):965–968, 1985.

29. Lovasz, L. On determinants, matchings, and random algorithms. In L. Budach, editor, *Fundamentals of Computing Theory*. Akademia-Verlag, Berlin, 1979.

30. Martel, C., and D. Gusfield. A fast parallel quicksort algorithm. *Information Processing Letters*, 30(2):97–102, 1989.

31. Meggido, N. Parallel algorithms for finding the maximum and the median almost surely in constant time. Technical report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1982.

32. Micali, S., and V. V. Vazirani. An $O(\sqrt{|V|}\,|E|)$ algorithm for finding maximum matching in general graphs. In *Proceedings of the ACM Symposium on Foundations of Computer Science*, Syracuse, NY, 1980, pp. 17–27.

33. Miller, G. L., and J. H. Reif. Parallel tree contraction and its applications. In *Proceedings Twenty-Sixth Annual IEEE Symposium on Foundations of Computer Science*, Portland, OR, 1985, pp. 478–489.

34. Mulmuley, K., U. V. Vazirani, and V. V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7(1):105–113, 1987.

35. Pan, V. Fast and efficient algorithms for the exact inversion of integer matrices. In *Proceedings Fifth Annual Foundations of Software Technology and Theoretical Computer Science Conference*, New Delhi, India, 1985, pp. 504–521.

36. Rabin, M. O. Probabilistic automata. *Information and Control*, 6(1-4):230–245, 1963.

37. Rabin, M. O. Probabilistic algorithms. In J. F. Traub, editor, *Algorithms and Complexity, Recent Results and New Directions*, Academic Press, Cambridge, MA, 1976.

38. Rabin, M. O., and V. V. Vazirani. Maximum matchings in general graphs through randomization. *Journal of Algorithms*, 10(4):557-567, 1989.

39. Raghaven, P. Lecture notes on randomized algorithms. Technical report, IBM Research Division, Yorktown Heights, NY, 1990.

40. Rajasekaran, S., and J. Reif. Derivation of randomized algorithms. Technical Report, Department of Computer Science, Harvard University, Cambridge, MA, 1984.

41. Rajasekaran, S., and J. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Computing*, 18(3):594-607, 1989.

42. Rajasekaran, S., and S. Sen. Random sampling techniques and parallel algorithm design. In J. H. Reif, editor, *Synthesis of Parallel Algorithms.* Morgan Kaufman, San Mateo, CA, 1991.

43. Reif, J. H. On the power of probabilistic choice in synchronous parallel computations. *SIAM J. Computing*, 13(1):496-503, 1984.

44. Reif, J. H., and S. Sen. Optimal randomized parallel algorithms for computational geometry. *Algorithmica*, in press, 1991.

45. Reischuk, R. Probabilistic parallel algorithms for sorting and selection. *SIAM J. Computing*, 14(2):396-409, 1985.

46. Rosser, J. B., and L. Schoenfeld. Approximate formulas for some functions of prime numbers. *Illinois J. Math*, 6:64-94, 1962.

47. Schwartz, J. Fast probabilistic algorithms for verification of polynomial identities. *JACM*, 27(4):701-717, 1980.

48. Sen, S. *Random Sampling Techniques for Efficient Parallel Algorithms in Computational Geometry.* PhD thesis. Department of Computer Science, Duke University, Durham, NC, 1989.

49. Solovay, R., and V. Strassen. A fast Monte-Carlo test for primality. *SIAM J. Computing*, 6(1):84-85, 1977.

50. Tamassia, R., and J. S. Vitter. Parallel transitive closure and point location in planar structures. *SIAM J. Computing*, 20(4):708-725, 1991.

51. Tutte, W. T. The factorization of linear graphs. *J. London Math. Soc.*, 22(1):107-111, 1947.

52. Valiant, L. G. A scheme for fast parallel communication. *SIAM J. Computing*, 11(2):350-361, 1982.

53. Valiant, L. G., and G. J. Brebner. Universal schemes for parallel communication. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, Milwaukee, WI, 1981, pp. 263-277.

54. Vazirani, V. Parallel graph matching. In J. H. Reif, editor, *Synthesis of Parallel Algorithms.* Morgan Kaufmann, San Mateo, CA, 1991.

55. Vishkin, U. Randomized speed-ups in parallel computation. In *Proceedings of the Sixteenth ACM Symposium on Theory of Computing*, Washington, DC, 1984, pp. 230-239.

10

*Limitations of PRAMs

In the previous chapters, we emphasized techniques and methods for designing efficient parallel algorithms. Almost all problems considered turned out to be highly parallelizable at an optimal or close to optimal cost. This chapter deals with the limitations of the PRAMs and with the ability of PRAMs to achieve dramatic speedups for all the problems in the class P, the class of problems solvable by polynomial-time sequential algorithms. There are three main implications of the results to be presented. First, we shall see that there is no significant difference in the relative powers of the different PRAM models used in this book. Second, there is an inherent limitation on the possible speed achievable for computing certain simple functions, even with a relatively large number of processors. Third, many important problems that can be solved with low-degree polynomial-time sequential algorithms are unlikely to be solvable extremely quickly on a CRCW PRAM with a polynomial number of processors.

We shall start, in Section 10.1, by presenting simulation results between the different PRAM models, followed by some lower-bound techniques for the CREW PRAM (Section 10.2) and for the EREW PRAM (Section 10.3). As a result, we show that the Boolean OR of n variables requires $\Omega(\log n)$ time on the CREW PRAM, *regardless* of the number of processors available (Section 10.2). In addition, we show that the problem of searching a sorted

list of n items requires $\Omega(\log n - \log p)$ time on a p-processor EREW PRAM, whereas it can be solved in $O(\frac{\log n}{\log p})$ time on the CREW PRAM (Section 10.3). Concerning lower bounds for the CRCW PRAMs, we take an indirect approach by first relating them to unbounded fan-in circuits (Section 10.4.1); we then use lower bounds developed for these circuits to deduce lower bounds for the CRCW PRAMs (Section 10.4.2). We finally discuss, in Section 10.5, elements of NC-theory as it relates to parallel computation, and we present a collection of important problems that we are unlikely to solve quickly with a polynomial number of processors.

10.1 Simulations Between PRAM Models

Different PRAM models have been used in the algorithms described in the previous chapters. Since our main goal has been to provide the simplest possible description of an algorithm, we have freely used any of the strong CRCW PRAM models whenever necessary. In this section, we explore the relationships among the different PRAM models. As a result, we shall see that a p-processor priority PRAM, our strongest PRAM model, can be simulated by a p-processor EREW PRAM, our weakest PRAM model, with a slowdown of $O(\log p)$ factor. Such a simulation cannot be improved, in general; in Sections 10.2 and 10.3, we shall provide examples for which such a gap must hold. The situation is more complicated when we investigate the relative powers of the different CRCW PRAMs. We shall shed light on these relationships later in this section.

10.1.1 SIMULATION OF THE PRIORITY CRCW PRAM ON THE EREW PRAM

Recall that the pipelined merge-sort algorithm (Algorithm 4.4) sorts n numbers in $O(\log n)$ time, using n processors. This algorithm can be adapted to run on the EREW PRAM. The pipelined merge-sort algorithm will form the basis of our first simulation result, given by the next theorem.

Theorem 10.1: *A concurrent read or write instruction of a p-processor priority CRCW PRAM can be implemented on a p-processor EREW PRAM to run in* $O(\log p)$ *time.*

Proof: We start by considering a concurrent-read instruction and its implementation on the EREW PRAM.

Let $Q_1, Q_2, \ldots, Q_p$ denote the processors of the priority CRCW such that processor Q_i has to read the content of location M_{j_i}. We designate by P_1, $P_2, \ldots, P_p$ the p processors of the simulating EREW PRAM. Then, processor P_i will try to simulate processor Q_i, where $1 \leq i \leq p$. A set of locations M_1, $M_2, \ldots, M_p$ of the global memory of the EREW PRAM is reserved for special use.

On the EREW PRAM, P_i sets up the pair $\langle j_i, i \rangle$ and stores the pair in the special location M_i. This step is a legal one on the EREW PRAM; it takes $O(1)$ time. Using the pipelined merge-sort algorithm, we sort the pairs $\langle j_i, i \rangle$, where $1 \leq i \leq p$, in lexicographically nondecreasing order. This process takes $O(\log p)$ time. The pairs $\langle j_i, i \rangle$ can now be viewed as organized in a number of blocks such that the pairs in each block have the same first component (which is the address of a global memory location). A representative of each block whose second component is minimum can be chosen in $O(1)$ time, since a comparison of the pair with the previous pair (whenever it exists) suffices to determine such a representative. Processors P_i's can then execute a concurrent read involving the locations indicated by the representative pairs on the EREW PRAM in $O(1)$ time. A segmented prefix-sums algorithm (see Exercise 2.5) can then be used to distribute each piece of data to its appropriate block in $O(\log p)$ time. Finally, each piece of data is stored in the special location indicated by its second component, which can then be accessed by the appropriate processor.

The handling of a concurrent write can be done in a similar fashion, except that we use triplets including the address, processor index, and the data to be written. □

Corollary 10.1: *Given an algorithm that can be implemented to run in time T on a p-processor priority CRCW PRAM, this algorithm can be implemented on a p-processor EREW PRAM to run in $O(T \log p)$ time.* □

Since any EREW algorithm will run unchanged on the CREW PRAM, the simulation also implies that the CREW PRAM can simulate the priority CRCW within the same bounds. We also have that the EREW PRAM can simulate the CREW PRAM within these bounds. These two simulation results cannot be improved in general.

Consider, for example, the problem of computing the Boolean OR function of n variables on an n-processor PRAM. As we have seen, this computation can be done in $O(1)$ time on the common CRCW PRAM, and hence obviously can be done in that time on the priority CRCW PRAM. However, we shall prove, in Section 10.2, that $\Omega(\log n)$ time is required on the CREW PRAM *regardless* of the number of processors. Later, in Section 10.3,

we shall present a version of a search problem that can be solved in $O(1)$ time on the CREW PRAM, but that requires $\Omega(\log n)$ time on the EREW PRAM.

10.1.2 SIMULATION OF THE PRIORITY CRCW PRAM ON THE COMMON CRCW PRAM

The relationship among the variants of the CRCW PRAM is more subtle. It is clear that any algorithm designed for the common CRCW will run correctly on the arbitrary CRCW, and any algorithm designed for the arbitrary CRCW will run correctly on the priority CRCW. This fact suggests that the priority CRCW is more powerful than the arbitrary CRCW, which in turn seems to be more powerful than the common CRCW. If, however, we do not place any restrictions on the number of processors or the amount of shared memory available, then the three models are equivalent. More precisely, we shall show that a concurrent-write step of a p-processor priority CRCW can be simulated by $O(1)$ steps on a $O(p \log p)$-processor common CRCW, assuming no bound is placed on the shared memory. Before presenting the proof, we shed some light on the nature of the task involved.

Consider a concurrent-write step by a p-processor priority CRCW PRAM. Let $M_1, M_2, \ldots, M_k$ be the global memory locations referred to in such a step. Then, for each M_i, the lowest-indexed processor wishing to write into M_i succeeds. A natural way to simulate such a step on another machine is to resolve the conflicts arising at each memory location separately, using a special reserved block of the shared memory. Each such problem can be rephrased as follows.

Leftmost Prisoner. Given a set of p processors, each having the value "live" or "dead" in its local memory, assign either a label $l_i = 1$ or $l_i = 0$ to each live processor P_i such that $l_i = 1$ if and only if P_i is the lowest-indexed (or leftmost) live processor. No processor with the value dead can participate in this computation. The size of the problem is defined to be p.

The leftmost prisoner problem seems to capture the essence of simulating a concurrent-write step of the priority CRCW. Note that this problem can be solved in $O(1)$ time on the common CRCW *if* we allow all the p processors to participate in the computation (see Exercise 2.13). Given our intention of using the "dead" processors to solve conflicts arising at other shared memory locations, no such processor can participate in this computation. However, this observation seems to indicate that a concurrent-write instruction on the priority CRCW can be simulated in $O(1)$ time on the common CRCW if we are willing to increase the number of processors available to the simulating machine. Such a fact is shown in the next lemma.

Lemma 10.1: *The leftmost-prisoner problem of size p can be solved in $O(1)$ time on the common CRCW PRAM if each live processor has $\lceil \log p \rceil$ auxiliary processors.*

Proof: Let $P_{i,1}, P_{i,2}, \ldots, P_{i,h}$ be the auxiliary processors associated with the live processor P_i, where $h = \lceil \log n \rceil - 1$, and set $P_{i,0} = P_i$. The goal of the processors $P_{i,j}$, where $0 \leq j \leq h$, is to determine whether or not P_i is leftmost among the live processors. We will make this determination by interacting with the auxiliary processors associated with other live processors. This interaction will be carried out through a complete binary tree T whose leaves are labeled by the indices of all the processors defining the leftmost-prisoner problem.

Let $v_{i,0}$ be the leaf associated with the live processor $P_{i,0}$ (which is the same as P_i). To fix notation, let us define a node $v_{i,j}$ inductively as follows. Node $v_{i,1}$ is the parent of $v_{i,0}$, and $v_{i,j}$ is the parent of $v_{i,j-1}$. The sibling of the node $v_{i,j}$ will be denoted by $sib(v_{i,j})$. Notice that, for $i \neq i'$ and $j \neq j'$, $v_{i,j}$ and $v_{i',j'}$ may denote the same node of T. It is easy to check that the label l_i of P_i should be set to 0 (that is, P_i is not leftmost) if and only if $v_{i,0}$ has an ancestor node $v_{i,j}$ such that $v_{i,j}$ is a right child and the subtree associated with $v_{i,j}$'s sibling contains a live processor.

The purpose of the auxiliary processor $P_{i,j}$ is to set a tag at $v_{i,j}$—say, $T(v_{i,j}) = 1$—indicating that the subtree rooted at $v_{i,j}$ contains a live processor. A concurrent write of the same value may occur at several nodes of T, a situation that is allowed in the common CRCW PRAM. It follows that $l_i = 0$ if and only if there is a node $v_{i,j}$ such that $v_{i,j}$ is a right child and $T(sib(v_{i,j})) = 1$. Processors $P_{i,j}$ can check this condition in $O(1)$ time, and therefore the labels l_i's can be set appropriately for all the live processors in $O(1)$ time. □

Lemma 10.1 immediately leads to the following theorem.

Theorem 10.2: *Any algorithm running on a p-processor priority CRCW PRAM in time T can be simulated on a $(p \log p)$-processor common CRCW PRAM in $O(T)$ time.*

Proof: With each processor Q_i of the priority CRCW PRAM associate $\lceil \log p \rceil$ processors $P_{i,j}$ of the common CRCW PRAM, where $0 \leq j \leq \lceil \log p \rceil - 1$. For each shared memory location of the priority CRCW, allocate a block of shared memory of size $O(p)$. Then, we can solve simultaneously all the leftmost-prisoner problems associated with the different memory locations that are concurrently written into. Lemma 10.1 shows that this computation can be done in $O(1)$ time. □

Although the common CRCW can simulate the priority CRCW with only a constant slowdown factor, the simulation is not efficient since the number of processors has to increase by a factor of $\log p$. In addition, the amount of shared memory has increased from, say, m global locations on the priority CRCW, to $\Theta(mp)$ on the common CRCW. It is possible to separate the various CRCW PRAMs if we restrict the number of processors and the amount of shared memory. Such restrictions will be considered in Exercises 10.6 through 10.8. Here, we shall present simulations in the case when the two involved machines have the same number of processors, but we place no restriction on the amount of shared memory used.

Simulation of the Priority CRCW on the Common CRCW Using the Same Number of Processors. We first simulate the priority CRCW on the common CRCW PRAM, where each has the same number p of processors. Theorem 10.1 indicates an upper bound of $O(\log p)$. We improve this bound in the next theorem.

Theorem 10.3: *A concurrent-write instruction of a p-processor priority CRCW PRAM can be simulated on a p-processor common CRCW PRAM in* $O\left(\frac{\log p}{\log \log p}\right)$ *time.*

Proof: To simplify the presentation, we shall view the operation of a CRCW PRAM as the synchronous execution of steps such that each step consists of three phases. In the first phase, each processor can perform a constant amount of local computation. In the second phase, a processor may write into a shared memory. In the third phase, a processor may read from the shared memory. We prove by induction the following claim.

Claim: The leftmost prisoner of $p = (t + 1)!$ processors can be solved in t steps on the common CRCW with a shared memory of size $m_t = \sum_{i=2}^{t+1} \frac{(t+1)!}{i!} < (t + 1)!$.

Proof of the Claim: We begin with the base case $t = 1$. The leftmost prisoner of two processors P_1 and P_2 can be solved in one step using a single shared memory location M. During the write (second) phase of the single step, P_1 writes 1 into M (if P_1 is live). During the read (third) phase, P_2 reads the content of M. Processor P_1 gets assigned the label $l_1 = 1$ (leftmost) if and only if P_1 is live. Processor P_2 gets assigned the label $l_2 = 1$ if and only if it is live and it reads 0.

Assume that the induction hypothesis holds for $p = t!$ processors. Let A be an algorithm solving the leftmost prisoner of $p = t!$ processors in $t - 1$

steps, using m_{t-1} shared memory locations. We shall show that the induction hypothesis holds for $p = (t + 1)!$ processors.

Divide the $(t + 1)!$ processors into $t + 1$ equal groups, where the ith group consists of processors $P_{t!(i-1)+1}$ through $P_{t!i}$, inclusive, and associate a block of memory of size m_{t-1} with each group. Since $m_t = (t + 1)m_{t-1} + 1$, the extra remaining location—say, M—will be used for the interactions among the different groups of processors.

Each group will perform $t - 1$ steps of algorithm A using its own block of memory. These steps, however, are scheduled such that they allow the proper interaction among the different groups through location M. Each group performs t steps such that $t - 1$ of these steps follow from algorithm A. During the ith step, the write and the read phases are special for the ith and the $(i + 1)$st groups. The write phase consists of all live processors of the ith group writing into location M; the read phase consists of all live processors of the $(i + 1)$st group reading the content of M. Otherwise, the steps are identical to those of algorithm A. Hence, each group performs at most a read phase and the next write phase through the global location M. It is clear that by the end of the ith step, all live processors in the $(i + 1)$st group will know whether there is a live processor of index lower than their indices; if there is, then their labels are determined immediately. If there is no live processor of lower index, algorithm A will guarantee the determination of the lowest-indexed live processor within the group, which is the lowest-indexed live processor, in this case.

Note that $e = \sum_{i \geq 0} \frac{1}{i!}$, and hence $e - 2 = \sum_{i \geq 2} \frac{1}{i!}$. This equality implies that $m_t \leq (e - 2)(t + 1)! < (t + 1)!$, and the claim is established.

The theorem follows Claim and Stirling's approximation, $t! \approx \frac{t^t}{e^t}\sqrt{2\pi t}$. □

10.1.3 SIMULATION OF THE PRIORITY CRCW PRAM ON THE ARBITRARY CRCW PRAM

Our final simulation of the priority CRCW on the arbitrary CRCW, where each has exactly p processors. Surprisingly, each step of the priority CRCW requires at most $O(\log \log p)$ steps on the arbitrary CRCW, as we show next.

Theorem 10.4: *A concurrent-write instruction of a p-processor priority CRCW PRAM can be simulated in* $O(\log \log p)$ *time on a p-processor arbitrary CRCW PRAM.*

Proof: We show how to solve the leftmost-prisoner problem for p processors on the arbitrary CRCW in $O(\log \log p)$ time, from which the theorem follows.

Assume, without loss of generality, that $\sqrt{p}$ is an integer. We partition the p processors of the arbitrary CRCW into $\sqrt{p}$ groups G_j, where $0 \leq j \leq \sqrt{p} - 1$, such that G_j contains processors $P_{j\sqrt{p}+1}$ through $P_{(j+1)\sqrt{p}}$, inclusive. We reserve a specific shared-memory location for each group—say, M_j for group G_j. For each group G_j, all the live processors attempt to write their indices into location M_j. In each group containing at least one live processor, a processor will be selected as the winner. If there is a winner in group G_j, we assign to it the new index j, and we recursively solve the leftmost-prisoner problem for $\sqrt{p}$ processors, where the winners are the live processors. Simultaneously, we solve the leftmost-prisoner problem induced within each group except the live processor selected as winner, if any. At the end of these computations, all the live processors in each group will know whether or not the lowest-indexed live processor is among them, and, in addition, the lowest-indexed live processor in each group will be easy to determine. This result is clearly sufficient to solve the initial leftmost-prisoner problem of p processors. The running time of this procedure satisfies the recurrence relation $T(p) = T(\sqrt{p}) + O(1)$, and therefore $T(p) = O(\log \log p)$. □

We turn next to proving lower bounds for specific problems on the CREW PRAM.

10.2 Lower Bounds for the CREW PRAM

Lower-bound arguments require a clean framework in which precise statements about the progress of a computation can be made. For example, we used the parallel comparison-tree model, in Chapter 4, to develop nontrivial lower bounds on the parallel complexity of several comparison problems. An adversary was capable of slowing down the progress of such computations by forcing the solution to lie in a large unexplored subset of the input. In this section, we introduce a strong version of the PRAM model, called the ideal PRAM model; we then use it to develop optimal lower bounds for elementary computations, such as computing the Boolean OR of n variables or computing the maximum of n elements.

10.2.1 THE IDEAL PRAM

An **ideal PRAM** consists of a number of processors $P_1, P_2, \cdots P_n$ with access to a global (shared) memory unit of *unbounded size*. Each processor is identical to a RAM together with a private memory unit of *unbounded size*. A computation consists of a number T of steps such that each step involves three

phases. During the **read phase,** a processor may read the content of a cell of the global memory. During the **compute phase,** a processor may perform *any amount of local computations* (including reading or writing from or into its local memory). During the **write phase,** a processor may write a value into a global memory location.

The rules regarding concurrent reads or concurrent writes depend on the particular version of the PRAM model used. Notice that an arbitrary amount of local computations are allowed in each step, that no limits are placed on the capacity of each location of the global or local memory, and that no limits are placed on the sizes of the global and local memories. A lower bound for the ideal PRAM reflects the *limitations of the communication scheme* used by the PRAM, since an unbounded amount of computations can be done at a single step. Obviously, lower bounds developed for the ideal PRAM model hold for our standard PRAM model (the converse is not true, however; see Exercise 10.15).

In this section, we shall establish the lower bound $\Omega(\log n)$ on the time required to compute the Boolean OR of n variables on the ideal CREW PRAM with *any* number of processors. This lower bound implies that almost any nontrivial problem requires logarithmic time on the CREW PRAM. In addition, it implies that the CREW PRAM is strictly weaker than the common CRCW PRAM, which can compute the Boolean OR function in $O(1)$ time, using n processors.

It may seem, at first glance, that $\log_2 n$ steps are "obviously" necessary to compute the Boolean OR of n variables on the CREW PRAM as implied by the balanced binary tree on n elements. However, it is possible to compute this function in $< \log_2 n$ steps by using the algorithm given in Exercise 10.14. A close examination of the algorithm reveals the interesting fact that we can communicate information by *not* writing into the global memory. Hence, a formal setup is needed to establish the desired lower bound. We start with a few definitions.

10.2.2 DEFINITIONS

Let $f(x_1, x_2, \ldots, x_n)$ be a Boolean function with n inputs. We shall use the notation $I = x_1 x_2 \cdots x_n$ to denote the input $(x_1, x_2, \ldots, x_n)$, and $I(i)$ to denote the input I whose ith component is complemented. An input I is **critical** for f if and only if $f(I) \neq f(I(i))$, for all $1 \leq i \leq n$.

EXAMPLE 10.1:

Let $f(x_1, x_2, \ldots, x_n) = x_1 + x_2 + \cdots + x_n$, where $+$ is the Boolean OR. Then, $I = 00 \cdots 0$ is a critical input for f. □

Consider the computation of the Boolean function $f(x_1, x_2, \ldots, x_n)$ on the ideal PRAM. Assume that the input values of $x_1, x_2, \ldots, x_n$ are stored initially in locations $M_1, M_2, \ldots, M_n$ of the global memory, and that, at the end of the computation, the value of the function f is to be stored in location M_1. If f has a critical input I, then, for any i, where $1 \leq i \leq n$, the value of the output is complemented when the value of x_i is complemented in the input I. Intuitively, this property means that the computation has to run long enough to allow the value of every x_i to affect the content of location M_1 appropriately. The exclusive-write assumption is crucial here since the effect of the values of all the x_i's cannot be transmitted to M_1 quickly. A more formal definition incorporating such a notion is given next.

An input index i **affects a memory location** M at time t on some input I if the content of M at time t with the input I differs from the content of M at time t with the input $I(i)$. In other words, the contents of M at time t are different on the two inputs I and $I(i)$. Define the set $L(M, t, I)$ as follows:

$$L(M, t, I) = \{i \mid i \text{ affects } M \text{ at time } t \text{ on input } I\}.$$

Similarly, we say that index i **affects processor** P at time t with input I if the state of P at time t on input I is different from the state of P at time t on input $I(i)$. The state of a processor P can be identified by the contents of all P's local registers and P's private memory. After the read phase of a step, the state of P may change. The new state will then determine the location written into and the value written by P during the write phase. The set $K(P, t, I)$ will denote the set of those input indices that affect P at time t with input I.

10.2.3 CHARACTERIZATION OF THE SETS $K(P, t, I)$ AND $L(M, t, I)$

The next two lemmas shed light on the nature of the two sets $K(P, t, I)$ and $L(M, t, I)$.

Lemma 10.2: *If $i \in K(P, t, I)$, where $t > 1$, then one of the following two cases must hold:*

1. $i \in K(P, t-1, I)$.
2. *P reads a global memory location M on input I at time t, and $i \in L(M, t-1, I)$.*

Proof: Let $i \in K(P, t, I)$. That is, the state of P on input I at time t is different from the state of P on input $I(i)$ at time t. Assume that $i \notin K(P, t-1, I)$. Hence, the state of P at time $t-1$ is the same, regardless of whether the input is I or $I(i)$. The only way for the state of P to become different at time t is for P to read

a global memory location M, and the contents of M are different on the inputs I and $I(i)$. This fact implies that $i \in L(M, t-1, I)$, where M is a global memory location read by P on input I at time t. □

Lemma 10.3: *If $i \in L(M, t, I)$, where $t > 1$, then one of the following two cases must hold:*

1. *A processor P writes into M at time t on input I, and $i \in K(P, t, I)$.*
2. *No processor writes into M at time t on input I, and either $i \in L(M, t-1, I)$ or a processor P writes into M at time t on input $I(i)$.* □

The proof of Lemma 10.3 is simple, and will be left to Exercise 10.11.

10.2.4 THE KEY FACT

We are now ready to state the following lemma, which gives the key fact needed to establish our lower bounds for the CREW PRAM.

Lemma 10.4: *Let P be a processor of a CREW PRAM, and let M be a location of the shared memory. Then, after t steps of an arbitrary computation, we have $|L(M, t, I)| \leq b^t$, and $|K(P, t, I)| \leq b^t$, where $b = \frac{1}{2}(5 + \sqrt{21})$.*

Proof: Let $k_0 = 0$, $l_0 = 1$, and let the sequences k_i and l_i be defined by $k_{t+1} = k_t + l_t$, $l_{t+1} = 3k_t + 4l_t$. We prove by induction the following claim.

Claim: $|K(P, t, I)| \leq k_t$ and $|L(M, t, I)| \leq l_t$, for any $t \geq 0$.

Proof of the Claim: We start with the base case $t = 0$. No index can affect P before the first step, and hence $|K(P, 0, I)| = 0$. On the other hand, at most one index can affect any memory location (if an input is stored in that location), and hence $|L(M, 0, I)| \leq 1$.

Suppose that the induction hypothesis holds for $K(P, t, I)$ and $L(M, t, I)$, $t \geq 0$. We show that the claim holds for $K(P, t+1, I)$ and $L(M, t+1, I)$.

Lemma 10.2 implies that $K(P, t+1, I) \subseteq K(P, t, I) \cup L(M, t, I)$, where M is the global memory location read by P on input I at time t. Therefore, the inductive claim for $K(P, t+1, I)$ follows.

To get the bound on $L(M, t+1, I)$, we use Lemma 10.3. Hence, we distinguish between the following two cases:

1. A processor P writes into location M on input I at time $t+1$. Thus, $|L(M, t+1, I)| \leq |K(P, t+1, I)| \leq k_t + l_t < l_{t+1}$.

2. No processor writes into M at time $t + 1$. An index i affects M at $t + 1$ if and only if either i affects M at t or a processor P writes into M at time $t + 1$ on input $I(i)$. In the latter case, we say that index i *causes* P to write into location M at $t + 1$ with input I. Hence, $L(M, t + 1, I) \subseteq L(M, t, I) \cup Y(M, t + 1, I)$, where $Y(M, t + 1, I)$ is the set of those indices that cause some processor P to write into M at time $t + 1$ with input I. Assume that index u_i causes processor P_{w_i} to write into M at time $t + 1$ with input I, where $1 \leq i \leq r$ for some integer r. That is, P_{w_i} writes into location M at time $t + 1$ on the input $I(u_i)$. We first note the following fact (see Fig. 10.1).

Fact: For all pairs u_i, u_j such that $P_{w_i} \neq P_{w_j}$, either $u_i \in K(P_{w_j}, t + 1, I(u_j))$ or $u_j \in K(P_{w_i}, t + 1, I(u_i))$.

The above fact is derived from the observation that if neither condition holds, then on input $I(u_i)(u_j)$, both P_{w_i} and P_{w_j} will write into location M at time $t + 1$, which is not allowed on the CREW PRAM model.

To finish the proof of the claim, we need to establish a bound on r, the number of indices that can cause some processor to write into M at time $t + 1$ with input I. Construct the bipartite graph consisting of the sets of vertices, $U = \{u_1, u_2, \ldots, u_r\}$ and $V = \{(I(u_1), P_{w_1}), (I(u_2), P_{w_2}), \ldots, (I(u_r), P_{w_r})\}$, such that u_i is connected to $(I(u_j), P_{w_j})$ if and only if u_i affects P_{w_j} at time $t + 1$ with input $I(u_j)$. To get the desired bound on r, we derive a lower and an upper bound on the total number e of edges in the bipartite graph.

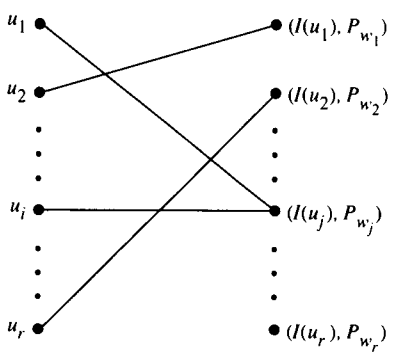


FIGURE 10.1
The bipartite graph for the proof of Lemma 10.4. u_i is connected to $(I(u_j), P_{w_j})$ if and only if u_i affects P_{w_j} at time $t + 1$ with input $I(u_j)$.

For each vertex $(I(u_j), P_{w_j})$, we have at most $|K(P_{w_j}, t+1, I(u_j))| \leq k_{t+1}$ edges (by the construction of the bipartite graph). Thus, $e \leq rk_{t+1}$. The lower bound on e can be obtained as follows. Let $\langle u_i, u_j \rangle$ be a pair such that $P_{w_i} \neq P_{w_j}$ (as outlined in the Fact). There are r choices of u_i, and, for each u_i, there are at most $|K(P_{w_j}, t+1, I)| \leq k_{t+1}$ indices u_j such that $P_{w_i} = P_{w_j}$. Hence, the total number of pairs $\langle u_i, u_j \rangle$ such that $P_{w_i} \neq P_{w_j}$ is at least $r(r - k_{t+1})$. By using the above Fact, we obtain $e \geq \frac{1}{2}r(r - k_{t+1})$. Combining this inequality with $e \leq rk_{t+1}$, we obtain that $r \leq 3k_{t+1} = 3k_t + 4l_t$. But $|L(M, t+1, I)| \leq |L(M, t, I)| + r \leq 3k_t + 4l_t = l_{t+1}$, and the proof of the claim follows.

We can now complete the proof of the lemma by establishing bounds on k_t and l_t. We can verify that the recurrences defining k_t and l_t have the following solutions:

$$k_t = \frac{b^t}{\sqrt{21}} - \frac{\bar{b}^t}{\sqrt{21}}$$

$$l_t = \frac{3+\sqrt{21}}{2\sqrt{21}} b^t + \frac{-3+\sqrt{21}}{2\sqrt{21}} \bar{b}^t,$$

where $b = \frac{1}{2}(5 + \sqrt{21})$ and $\bar{b} = \frac{1}{2}(5 - \sqrt{21})$. Therefore, $k_t \leq b^t$ and $l_t \leq b^t$, for the stated b value. □

10.2.5 LOWER BOUNDS FOR SPECIFIC PROBLEMS

We have just developed the necessary tools to establish our main lower-bound theorem for this section.

Theorem 10.5: *Let f:$\{0, 1\}^n \rightarrow \{0, 1\}$ have a critical input. Then, $\Omega(\log n)$ steps are required to compute f on a CREW PRAM with any number of processors.*

Proof: According to our conventions, the output value should appear in location M_1 at the end of the computation. Hence, n indices must affect location M_1; that is, $|L(M_1, T, I)| \geq n$. Using Lemma 10.4, we obtain that $b^T \geq |L(M, T, I)| \geq n$, and hence the theorem follows. □

Using the fact that the OR function has a critical input, we get the following corollary.

Corollary 10.2: *A CREW PRAM requires $\Omega(\log n)$ time to compute the Boolean OR of n variables, regardless of the number of processors available.* □

The proof of the following corollary involves simple reductions from the problem of computing the Boolean OR of n inputs to each of the problems listed. Its proof is left as an exercise.

Corollary 10.3: *Each of the following problems requires* $\Omega(\log n)$ *time on the CREW PRAM with any number of processors.*

1. *Sorting a sequence* $x_1, x_2, \ldots, x_n$, *where* $x_i \in \{0, 1\}$
2. *Computing the sum* $x_1 + x_2 + \cdots + x_n$, *where each* $x_i \in \{0, 1\}$
3. *Computing the maximum function on n inputs* □

We next extend the techniques developed in this section to obtain an interesting lower bound for the EREW PRAM.

10.3 Lower Bounds for the EREW PRAM

We have shown, in Section 10.2, that the problem of computing the Boolean OR of n variables requires $\Omega(\log n)$ time on the CREW PRAM, regardless of the number of processors available. This result implies that the weakest CRCW PRAM is strictly more powerful than the CREW PRAM. In this section, we will show that the CREW PRAM is *strictly* more powerful than the EREW PRAM. The search problem considered in Chapter 4 will serve as a vehicle for establishing this fact. The proof techniques used are similar to those introduced in Section 10.2.

10.3.1 A SEARCH PROBLEM

Let $X = (x_1, x_2, \ldots, x_n)$ be a sequence of n distinct elements drawn from a linearly ordered set $(S, \leq)$ such that $x_1 < x_2 < \cdots < x_n$. The search problem is to determine, for a given element $y \in S$, the index i for which $x_i \leq y < x_{i+1}$, where $x_0 = -\infty$ and $x_{n+1} = +\infty$ are added to S. Earlier, we described a simple CREW PRAM algorithm (Algorithm 4.1) to solve this problem in $O\left(\frac{\log (n+1)}{\log (p+1)}\right)$, where p is the number of processors used. An EREW PRAM algorithm achieving the bound $O(\log n - \log p)$ time is straightforward, assuming that the search key is made available to all the processors. We decompose the input array into p blocks, each of size $\approx \frac{n}{p}$. Each processor performs a binary search on a block in $O\left(\log \left(\frac{n}{p}\right)\right) = O(\log n - \log p)$ time. The successful processor will store the result in the appropriate location of the global memory.

In this section, we shall show that an EREW PRAM requires $\Omega(\log n - \log p)$ time, even if the search key y is initially made available to all the

processors. This result implies that, for $p = \Theta(n^{\alpha})$, where α is any constant such that $0 < \alpha < 1$, the search problem requires $\Omega(\log n)$ time on the EREW PRAM, whereas it can be solved in $O(1)$ time on the CREW PRAM (Algorithm 4.1). We therefore conclude that the CREW PRAM is strictly more powerful than the EREW PRAM.

Since the search key will be assumed to be known to all the p processors of the EREW PRAM, our search problem can be reformulated as the following **zero-counting problem.** Given a *monotonic binary* sequence $x_1, x_2, \ldots, x_n$, determine the index i such that $x_i = 0$ and $x_{i+1} = 1$. We interpret $x_i = 0$ to mean a value smaller than the search key, and $x_i = 1$ to mean a value greater than the search key.

As before, we assume the ideal PRAM model, and further assume that the input values of $x_1, x_2, \ldots, x_n$ are stored initially in locations $M_1, M_2, \ldots, M_n$. At the end of the computation, we will indicate the output i by changing the content of location M_i' of a set of n reserved output locations M_j', where $1 \leq j \leq n$.

10.3.2 INDICES AFFECTING A PROCESSOR OR A LOCATION

Let I_i denote the input consisting of i zeros followed by $n - i$ ones. Index i **affects location** M at step t if the content of M at step t on input I_i is different from the content of M at step t on input I_{i-1}. We define $L(M, t)$ to be the set $\{i \mid i \text{ affect } M \text{ at step } t\}$.

Similarly, index i **affects processor** P at step t if the state of P at step t on input I_i is different from the state of P at step t on input I_{i-1}. We define $K(P, t)$ to be the set $\{i \mid i \text{ affects } P \text{ at step } t\}$.

The next two lemmas, which are similar to Lemmas 10.2 and 10.3 of Section 10.2, characterize the indices that belong to $L(M, t)$ and $K(P, t)$.

Lemma 10.5: *If $i \in K(P, t)$, where $t > 1$, then one of the following two cases must hold:*

1. $i \in K(P, t-1)$.
2. *P reads a certain location M on input I_i at step t, and $i \in L(M, t-1)$.*□

The proof of Lemma 10.5 is similar to the proof of Lemma 10.2.

Lemma 10.6: *If $i \in L(M, t)$, where $t > 1$, then one of the following three cases must hold:*

1. $i \in L(M, t-1)$.

2. *There is a processor P that writes into M at step t on input I_i, and $i \in K(P, t)$.*
3. *There is a processor P that writes into M at step t on input I_{i-1}, and $i \in K(P, t)$.*

Proof: Suppose that $i \notin L(M, t-1)$. Hence, the content of M at step $t-1$ is the same, regardless of whether the input is I_{i-1} or I_i. It follows that the content of M is modified during step t. The only way that the content of M can be changed at step t is for some processor P to write into M. Suppose that P writes into M at step t on input I_i. Then, we must have $i \in K(P, t)$, because otherwise the state of P at step t is the same for the two inputs I_i and I_{i-1}, and therefore the same value will be written into M. Similarly, a processor P' may write into location M at step t on input I_{i-1}, in which case $i \in K(P', t)$. □

10.3.3 THE MAIN LOWER BOUND

We are ready to prove our main lower bound.

Theorem 10.6: *Given a monotonic binary sequence of length n, the problem of counting the number of zeros requires $\Omega(\log n - \log p)$ time on the EREW PRAM with p processors.*

Proof: Let $K(P,t)$ and $L(M,t)$ be as introduced in Section 10.3.2. The progress of the computation will measured by the function $c(t) = \sum_P |K(P, t)| + \sum_M \max\{0, |L(M,t)| - 1\}$.

Claim: $c(t) \leq 6c(t-1) + 3p$.

Proof of the Claim: We first use Lemma 10.6 to establish an upper bound on the sum $\sum_M |L(M,t)|$. Let $L'(M,t) = L(M,t) - L(M,t-1)$—that is, all the indices in $L(M,t)$ but not in $L(M,t-1)$. Then, according to Lemma 10.6, for each index $i \in L'(M,t)$, there is a processor P writing into M at time t on input I_i (or input I_{i-1}), and $i \in K(P,t)$. Since a processor may write into a single global location in a time step, each $i \in K(P,t)$ can contribute at most two occurrences in $L'(M,t)$ (one corresponding to writing on input I_i, and the other corresponding to writing on input I_{i-1}). Hence, $\sum_M |L'(M,t)| \leq 2\sum_P |K(P,t)|$, and thus

$$\sum_M |L(M,t)| \leq \sum_M |L(M,t-1)| + 2\sum_P |K(P,t)|$$

We can assume inductively that $L(M,t-1) \subseteq L(M,t)$, because otherwise we can replace $L(M,t)$ with $L(M,t) \cup L(M,t-1)$. Clearly, an upper bound for the resulting sum will also be an upper bound for the original sum. Hence, we have that $|L(M,t)| \geq |L(M,t-1)|$, which implies that

$$\sum_M \max\{0, |L(M,t)| - 1\} \leq \sum_M \max\{0, |L(M,t-1)| - 1\} + 2\sum_P |K(P,t)| \tag{10.1}$$

We now use Lemma 10.5 to establish an upper bound on the sum $\sum_P |K(P,t)|$. Let $K'(P,t) = K(P,t) - K(P,t-1)$. According to Lemma 10.5, for each index $i \in K'(P,t)$, there is a memory location M such that P reads M on input I_i and $i \in L(M,t-1)$. Since we are assuming the EREW PRAM model, no global memory location can be accessed by more than a single processor at any given time step. Hence, given M, each index $i \in L(M,t-1)$ can contribute at most one index in $K'(P,t)$ for some processor P.

Let $J(i,t)$ be the set of global locations accessed at time t on input I_i, and let $J_t = \cup_i J(i,t)$. Clearly, $\sum_P |K'(P,t)| \leq \sum_{M \in J_t} |L(M,t-1)|$. It follows that

$$\sum_P |K(P,t)| \leq \sum_P |K(P,t-1)| + \sum_{M \in J_t} |L(M,t-1)|.$$

An upper bound on $|J_t|$ can be obtained as follows. Given a processor P, the number of *distinct* locations accessed by P at time t is $\leq |K(P,t-1)| + 1$, and hence $|J_t| \leq \sum_P (|K(P,t-1)| + 1) = \sum_P |K(P,t-1)| + p$, where p is the total number of processors available. This observation implies that

$$\begin{aligned}
\sum_P |K(P,t)| &\leq \sum_P |K(P,t-1)| + \sum_{M \in J_t} |L(M,t-1)| \\
&= \sum_P |K(P,t-1)| + \sum_{M \in J_t} (|L(M,t-1)| - 1) + |J_t| \\
&\leq 2\sum_P |K(P,t-1)| + \sum_M \max\{0, |L(M,t-1)| - 1\} + p.
\end{aligned}$$

Combining the last inequality with the inequality in Eq. 10.1, we obtain

$$\sum_M \max\{0, |L(M,t)| - 1\} + 3\sum_P |K(P,t)| \leq 4\sum_M \max\{0, |L(M,t-1)| - 1\} + 2\sum_P |K(P,t)| + 6\sum_P |K(P,t-1)| + 3p.$$

Therefore, $c(t) \leq 6c(t-1) + 3p$, and the claim follows.

We can now complete the proof of the theorem by noting that $c(T) \geq n$, where T is the total number of steps required. Given our assumption on the form of the output, it is easy to verify that each of the reserved locations for the output will be affected by at least two indices. On the other hand, the inequality $c(t) \leq 6c(t-1) + 3p$, together with the initial condition $c(0) = 0$, implies that $c(t) \leq \left(\frac{6^t - 1}{5}\right) 3p$. Therefore, we must have $T = \Omega(\log n - \log p)$. □

We next tackle the problem of proving lower bounds for computing some specific simple functions on the CRCW PRAM model, our strongest parallel model.

10.4 Lower Bounds for the CRCW PRAM

In Sections 10.2 and 10.3, we showed optimal lower bounds on the parallel complexity to solve a few simple problems on the ideal PRAM under the CREW and the EREW assumptions. Lower bounds for the ideal PRAM reflect the limitations on the communication assumptions, since an unbounded amount of computations is allowed at each step. Establishing similar lower bounds for the CRCW PRAM appears to be substantially more difficult. Let us start by noting that the techniques used in the previous sections are not suitable for capturing the progress of a computation on the CRCW PRAM model, since bounding the number of processors that can "affect" a memory location at a given step is not possible under the concurrent-write assumption. In fact, a common CRCW can compute *any* Boolean function in $O(1)$ time, as the following theorem shows.

Theorem 10.7: *Let f:$\{0, 1\}^n \rightarrow \{0, 1\}$ be an arbitrary Boolean function. Then, f can be computed in $O(1)$ time on a common CRCW PRAM with $\leq n2^n$ processors.*

Proof: We write f as a sum of minterms $f = \sum_i m_i(x_1, x_2, \ldots, x_n)$, where each m_i is a product of n literals, a literal being a variable or that variable's complement. The number of minterms is at most 2^n. With each term m_i, we associate n processors P_{ij}, for $1 \leq j \leq n$. The goal of processors P_{ij} is to compute the term m_i, which is the Boolean AND of n literals. Since the AND of n variables can be computed in $O(1)$ time, using n processors, on the common CRCW, $n2^n$ processors can compute all the m_i's simultaneously in $O(1)$ time. To compute f, we can compute the Boolean sum of the m_i's in $O(1)$ time, using the processors P_{i1}, for all i. Therefore, f can be computed in $O(1)$ time, using at most $n2^n$ processors on the common CRCW. □

Theorem 10.7 illustrates the fact that, with the concurrent-write capability, many processors can cooperate effectively to compute a function. In addition, we have already encountered simple problems that can be solved in $O(1)$ time on a CRCW PRAM with a *polynomial* number of processors. In particular, we know how to compute the maximum of n elements, and how to solve the problem of all nearest smaller values (a generalization of the merging problem, see Exercise 2.14), in $O(1)$ time, using n^2 processors. These observations indicate that it would be interesting to identify simple computations that cannot be carried out even in constant time by a CRCW PRAM, using a polynomial number of processors. There are such simple functions, such as the parity function, but the known proofs are highly nontrivial and require the introduction of techniques initially developed for circuit complexity.

In this section, we shall actually prove a much stronger lower bound for computing the parity function. Before embarking on our endeavor, we state the result we seek.

Theorem 10.8: *The problem of computing the parity function of n variables requires $\Omega\left(\frac{\log n}{\log \log n}\right)$ time on the priority CRCW PRAM, using a polynomial number of processors.* □

We shall prove Theorem 10.8 under certain assumptions of the priority CRCW PRAM. Our proof strategy will consist of two main steps. The first step is to relate the CRCW PRAMs to the class of circuits allowing unbounded fan-in (to be introduced in the next section) by providing efficient simulations between the two models. The second step is to establish lower bounds for unbounded fan-in circuits computing the parity function.

10.4.1 THE RELATIONSHIP BETWEEN CRCW PRAMS AND UNBOUNDED FAN-IN CIRCUITS

A (Boolean) **circuit** is a directed acyclic graph such that each node of indegree greater than 0 is labeled as any one of an AND gate, an OR gate, or a NOT gate. The nodes of indegree 0 are called **inputs** and are labeled with a variable, or with a constant 0 or 1. Certain nodes are also designated as **output nodes.** The indegree and the outdegree of a node will be referred to as the **fan-in** and the **fan-out** of that node, respectively. A circuit represents a set of Boolean functions in a natural way.

The **size** of a circuit is the total number of edges in the circuit. The **depth** of a circuit is the length of the longest path between an input node and an output node.

EXAMPLE 10.2:

Let $f(x_1, x_2, x_3) = x_1'x_2'x_3 + x_1'x_2x_3' + x_1x_2'x_3' + x_1x_2x_3$ be the parity function on three variables. A circuit that computes $f(x_1, x_2, x_3)$ is shown in Fig. 10.2. □

Unbounded Fan-In Circuits. Of particular interest to us are the circuits with **unbounded fan-in**—that is, circuits for which an AND gate or an OR gate can have an arbitrary number of inputs. These circuits are also referred to as **bounded-depth circuits,** since they tend to have very small depths. In fact, we can trivially realize any Boolean function f by a circuit whose depth is ≤ 3 simply by using the disjunctive or the conjuctive normal form of f. The size of the circuit is at most $n2^n$, where n is the number of variables (compare with Theorem 10.7).

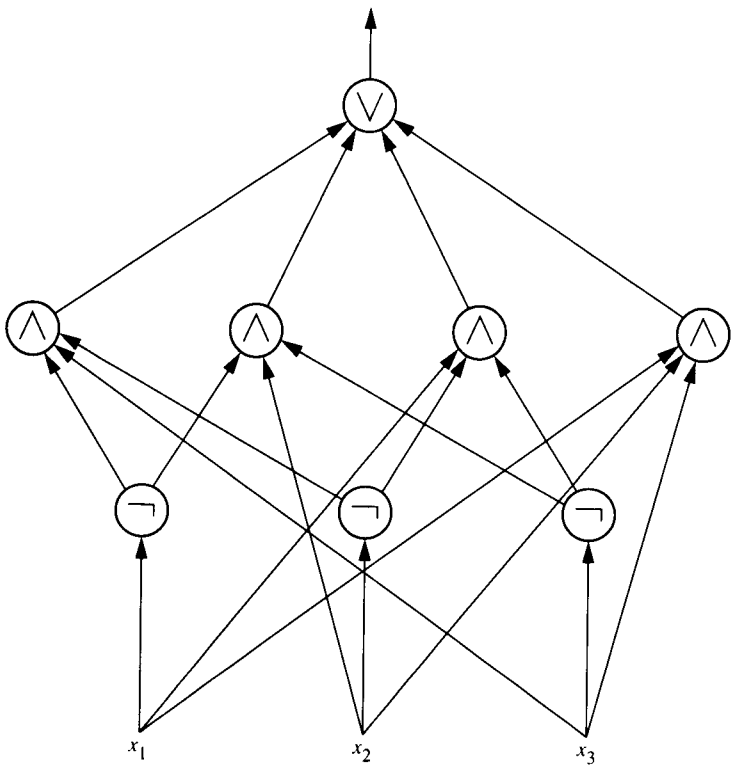


FIGURE 10.2
A circuit for the parity function of three variables.

A challenging task is to establish nontrivial lower bounds on the sizes of circuits performing certain computations as a function of their depths. One such result (to be shown later) is that the size of any constant-depth circuit computing the parity of n variables must be exponential. Unbounded fan-in circuits have provided an extremely useful tool to show important lower bounds with applications in various areas of complexity theory. Our interest in these circuits stems from their close relationship to CRCW PRAMs. Our next task is to elaborate on this relationship so as to be able to translate lower bounds from the circuit model into the CRCW PRAM model.

For the remainder of this section, we shall denote the Boolean operators AND and OR by the symbols $\wedge$ and $\vee$, respectively, whenever they appear in a Boolean expression.

Simulation of Circuits by CRCW PRAMs. We begin by describing a simple simulation of circuits by CRCW PRAMs.

Theorem 10.9: *Let $f:\{0, 1\}^n \rightarrow \{0, 1\}^m$ be a function of n inputs and $m \leq n$ outputs, and let C be a circuit that computes f. Suppose that C is of size $S(n)$ and depth $D(n)$. Then, f can be computed by a common CRCW PRAM in $O(D(n))$ time, using $S(n) + n$ processors.*

Proof: We use *DeMorgan's law* $\wedge_{1 \leq i \leq k} g_i = (\vee_{1 \leq i \leq k} g_i')'$ to replace each AND gate with a set of NOT and OR gates. Hence, we can assume that our circuit consists of OR and NOT gates only. The size and the depth of our circuit have each increased by a factor of at most 3; hence, this change will not affect our asymptotic bounds.

We number the nodes of C starting with the input nodes. We associate a global memory location M_i with node i. Initially, the jth input is stored in location M_j, where $1 \leq j \leq n$. The CRCW PRAM will simulate C, level by level, starting from the bottom level, such that the value generated at node i will be computed and stored at the global location M_i of the PRAM. More precisely, we allocate a processor P_{ij} with each edge from node i to node j. Suppose that node j is at depth d. Then, using a local counter, P_{ij} waits for cd steps, where c a constant large enough to allow the computations at lower depth to be carried out by the CRCW PRAM. Then, if node j is an *OR* gate, processor P_{ij} writes 1 in location M_j if location M_i contains 1; otherwise, P_{ij} stays idle. If node j is a *NOT* gate, then P_{ij} stores the complement of M_i into location M_j.

Hence, each level of the circuit takes $O(1)$ time to simulate on the PRAM, and thus $O(D)$ time is sufficient to reach the output nodes. In addition, m additional processors are assigned to the output nodes. After waiting for cD steps, these processors move the output bits to the appropriate locations of the global memory. □

Remark 10.1: Given a family of circuits $\{C_n\}$ to compute a Boolean function f on n inputs, it is important to be able to generate C_n efficiently for each n. Otherwise, it would be possible to design a different circuit for each n ("nonuniform" circuits). Let us first mention that nonuniform circuits can compute nonrecursive functions. In the preceding simulation, nonuniform circuits will generate different programs for different input sizes. All our PRAM algorithms, however, have been described for an arbitrary value of n.

Therefore, we make an assumption that will guarantee a simple description of C_n for each n. Typically, a notion of **uniformity** is introduced to deal with this problem. We shall adopt the following definition.

A circuit is **uniform** if, for each n, the circuit C_n can be described by a deterministic Turing machine in $O(\log n)$ space. We can give a specification

of a circuit by listing all the gates, and the type and the inputs of each gate. The space notion used here assumes that the input is stored in a read-only memory, which is not counted as a part of the space used. The fact that only $O(\log n)$ space is required to describe C_n indicates that C_n is indeed a simple circuit. It also implies that C_n can be generated in polynomial time. All our circuits will be assumed to be uniform. □

Our goal, for the remainder of this section, is to establish a converse to Theorem 10.9. More precisely, we would like to show how to simulate a CRCW PRAM with $P(n)$ processors and time bound $T(n)$ by a circuit whose size is polynomial in $P(n)$ and $T(n)$ and whose depth is $O(T(n))$. Such a simulation is not possible unless we put restriction on the word size and the instructions allowed in the CRCW PRAM. We shall next introduce into our PRAMs a few assumptions that will make it possible to obtain the desired simulation. These assumptions seem to be natural and not to be overly restrictive.

The Restricted PRAM. For our **restricted** PRAMs, each program can have instructions from the following list. The notation R_i is used to indicate a local memory register.

1. R_i: = *constant*; the effect is to load local register R_i with *constant*.
2. R_i: = R_j; the effect is to load the content of R_j into R_i.
3. R_i: = *PID*; the effect is to store processor ID into register R_i.
4. R_i: = $R_j \pm R_k$; the effect is to add or subtract the contents of the registers R_j and R_k and store the result in R_i.
5. R_i:= $*R_j[l, g]$; the effect is to use the content of register R_j as the address of a read instruction from the local or global memory, depending on whether l or g is used, respectively. The result is stored in register R_i.
6. $*R_i := R_j[l,g]$; the effect is to write the content of R_j into a local or global memory location whose address is the content of register R_i.
7. **go to** m **if** $R_i([<, =])R_j$; the effect is to transfer control to the instruction labeled by m if the condition is satisfied.
8. **halt**; the effect is to force the processor to stop.

Without loss of generality, we assume that no nonpositive addresses appear in a program.

An input of **size** n consists of n words, each requiring at most n bits. An input of size n is initially stored in the first n global locations of the PRAM. Each instruction takes one unit of time. The machine operates in time $T(n)$ if all processors halt within $T(n)$ steps on any input of size n.

Circuits for Addition and Comparison. Our simulation will require that the instructions executed at each time step be simulated by a constant-depth and small-sized circuit. Leaving aside for now the problems of obtaining the operands and storing the results, let us derive the circuits for performing addition and comparison. The circuit for subtraction is similar to that for addition. The circuits are described in the next two examples.

EXAMPLE 10.3: (Addition)

Consider the addition of two m-bit numbers $A = a_m \ldots a_2 a_1$ and $B = b_m \ldots b_2 b_1$. The look-ahead carry algorithm consists of computing the *carry generate* and the *carry propagate* bits given, respectively, by $g_i = a_i b_i$ and $p_i = a_i \oplus b_i$, for $1 \leq i \leq m$. Then, the ith carry bit c_i is given by $c_i = g_i + \sum_{1 \leq j \leq i-1} p_i p_{i-1} \ldots p_{j+1} g_j$, and the ith sum bit s_i is given by $s_i = p_i \oplus c_{i-1}$. Each c_i can be realized by a constant-depth circuit of size $O(m^2)$, as illustrated in Fig. 10.3. Therefore, all the carry bits can be computed by a constant-depth circuit of size $O(m^3)$. Generating the sum bits s_i will not increase the asymptotic bounds of the corresponding circuit. □

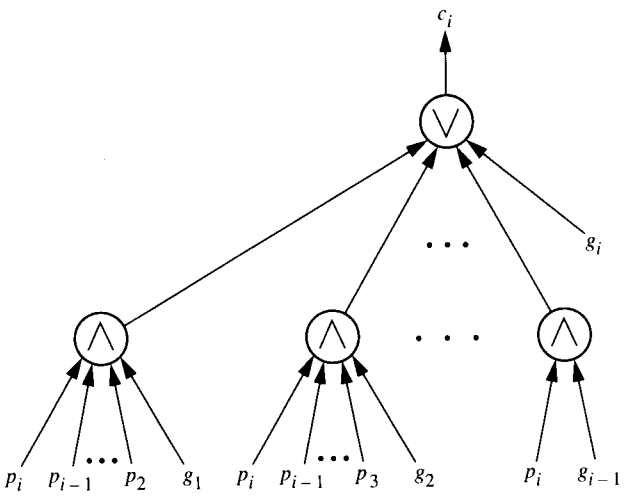


FIGURE 10.3
A constant-depth circuit to compute the ith carry bit resulting from adding two m-bit numbers a and b. The carry generate g_i and the carry propagate p_i are defined by $g_i = a_i b_i$ and $p_i = a_i \oplus b_i$.

EXAMPLE 10.4: (Comparison)

Given the two numbers $A = a_m \ldots a_2 a_1$ and $B = b_m \ldots b_2 b_1$, we would like to design a constant-depth and polynomial-sized circuit that generates the value 1 whenever $A < B$. Let $x_i = a_i b_i + a_i' b_i'$—that is, $x_i = 1$ if and only if $a_i = b_i$, for $1 \leq i \leq m$. Then, the output can be expressed by the Boolean function $a_m' b_m + x_m a_{m-1}' b_{m-1} + x_m x_{m-1} a_{m-2}' b_{m-2} + \ldots + x_m x_{m-1} \cdots x_2 a_1' b_1$. Clearly, such a Boolean function can be realized with a constant depth circuit of size $O(m^2)$. Figure 10.4 illustrates the corresponding circuit. □

Simulation of a Restricted CRCW PRAM by a Circuit. Let us try now to understand the problems involved in simulating a CRCW PRAM by a circuit.

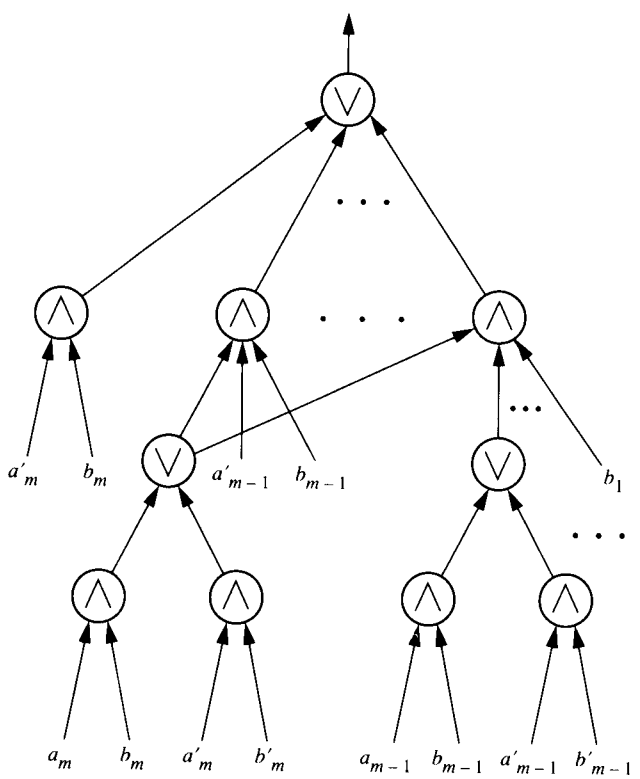


FIGURE 10.4
A constant-depth circuit to test whether $A < B$, where $A = a_m \ldots a_2 a_1$ and $B = b_m \ldots b_2 b_1$.

Let $P(n)$ be the number of processors on our CRCW PRAM, n being the input size. Recall that, by our earlier assumptions, the input consists of n words, each of length at most n bits, and that these words are initially stored in the first n global locations of the PRAM. Let $T(n)$ be the time bound of the PRAM algorithm. Thus, every processor halts within $T(n)$ steps on any input of size n.

Each processor of the PRAM will execute dynamically a sequence of instructions, each such instruction belonging to the set of instructions specified before. Contrast this behavior with that of a circuit that possesses a static structure realizing a fixed set of Boolean functions. To construct a simulating circuit, we have to specify a set of Boolean functions that can realize the input-output behavior of the PRAM algorithm.

A naive approach would be to derive the functions relating the input and the output of the algorithm by considering all possible values of the n input words. However, the size of the corresponding circuit may be exponential in n, which is not acceptable. What we shall do instead is to represent efficiently all possible values that can arise at a time step t, and to describe the Boolean functions relating these values as they change between steps t and $t + 1$. The circuit needed to realize such a transition will be of constant depth and will be of polynomial (in $P(n)$, $T(n)$, and n) size.

Our starting point concerns the representation of all possible values arising in an execution of the PRAM algorithm on an input of size n. These values are any one of the input values, or constants and addresses appearing in the programs of the processors, or values generated and stored by the processors. We now claim that the maximum number of bits needed to represent any of these values is given by $L = \max(n, \log P(n)) + T(n) + 1$. This claim follows from the fact that, initially, the length of any piece of data in the memory or the program is at most $\max(n, \log P(n))$, and, after each operation, the length increases by at most one bit. An extra bit is needed to distinguish between positive and negative numbers.

For each processor Q, we represent the local registers and their values by the triples $(a_l(Q,k,t), v_l(Q,k,t), w_l(Q,k,t))$, where $a_l(Q,k,t)$ and $v_l(Q,k,t)$ are L-bit words, and $w_l(Q,k,t)$ is a bit indicating a write operation. This triple should be interpreted as follows. If $w_l(Q,k,t) = 1$, the local register with address $a_l(Q,k,t)$ of processor Q contains the value $v_l(Q,k,t)$ at time t. If $w_l(Q,k,t) = 0$, the triple means nothing. A processor can write into at most one local register at each time step; hence, we can take $1 \le k \le T$. Note that these triples represent Boolean variables whose values are determined by the inputs to the PRAM.

In a similar fashion, we can represent the global memory locations by the triples $(a_g(k,t), v_g(k,t), w_g(k,t))$. In this case, the value of k satisfies $1 \le k \le n + P(n)T(n)$, since at most $P(n)$ values can be written into the global memory at each time step.

We now discuss the transformation on the memory triples that has to take place during the execution of step t of the PRAM. Such transformation depends on the instruction being executed; hence, the circuit corresponding to step t should contain subcircuits simulating the different types of PRAM instructions that might arise. The particular instruction executed by processor Q at time t will be represented by the Boolean variable $IC(Q, j, t)$, where $1 \leq j \leq s(Q)$ and $s(Q)$ is the number of instructions of Q's program. More precisely, $IC(Q, j, t) = 1$ if and only if Q executes the jth instruction at time t. The remaining details will be provided in the proof of the following theorem.

Theorem 10.10: *A restricted priority CRCW PRAM running in time $T(n)$ and using $P(n)$ processors can be simulated by a circuit of depth $O(T(n))$ and of size polynomial in $P(n)$, n, and $T(n)$.*

Proof: All addresses of memory locations (local or global) and the values stored in these locations will be represented by the triples as described just before the statement of the theorem.

Let $x_1, x_2, \ldots, x_n$ be the input stored in the first n locations of the global memory. Each x_i is padded out to L bits. At the start of the computation, we set $(a_g(k, 0), v_g(k, 0), w_g(k, 0)) = (k, x_k, 1)$, for all $1 \leq k \leq n$. All other memory triples are set to $(0, 0, 0)$ for $t = 0$. Moreover, we set $IC(Q, j, 0) = 0$ for $j \neq 1$, and $IC(Q, 1, 0) = 1$. This assignment of the IC values implies that the instruction counter of each processor is set to 1.

We will carry out the simulation by showing how to realize the changes to the memory triples and IC bits after each step of the CRCW PRAM by a circuit of constant depth and polynomial size. The inputs of each such step are the memory triples $(a_l(Q, k, t), v_l(Q, k, t), w_l(Q, k, t))$ for $1 \leq k \leq T$ and $1 \leq PID(Q) \leq P(n)$, $(a_g(k, t), v_g(k, t), w_g(k, t))$, for $1 \leq k \leq n + P(n)T(n)$ and the IC bits $IC(Q, j, t)$, for $1 \leq j \leq s(Q)$ and $1 \leq PID(Q) \leq P(n)$. The outputs are the updated values as required by the execution of step t of the PRAM algorithm.

A high-level description of the circuit that simulates one step of a processor is shown in Fig. 10.5. The main circuits required are those that realize the following tasks: (1) compute operands, (2) execute operation, (3) update IC values, (4) select result, and (5) update memory. We now consider each task separately.

We fix a processor Q and a time step t.

1. *Compute operands:* The outputs of this particular subcircuit are the values of the operands needed to execute the next instruction. The value of an operand is equal to the content of the local register referred to by the instruction being executed by Q at time t. Let $op_1(Q, j)$ be the L-bit representation of the first operand in the jth instruction of the program

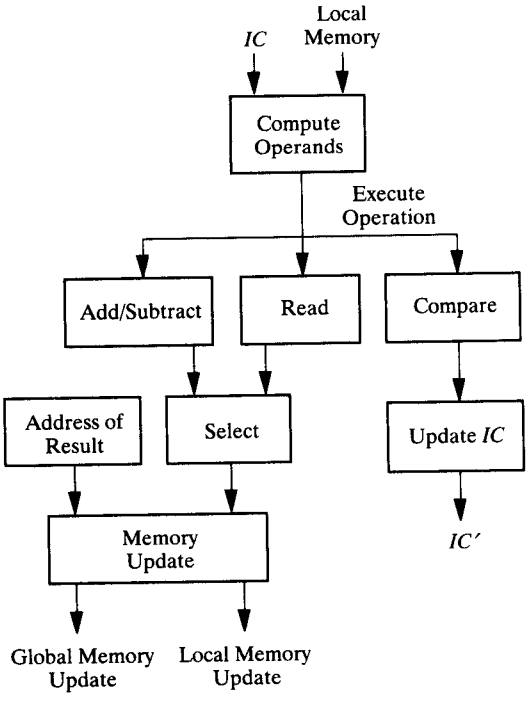


FIGURE 10.5
A high-level description of the circuit that simulates a step of a single processor of the PRAM.

of processor Q. If the instruction has no operand, then set $op_1(P, j) = 0$. The value of the first operand can be expressed as

$$\bigvee_{j=1}^{s(Q)} \bigvee_{k=1}^{T} IC(Q, j, t) \wedge EQ(op_1(Q, j), a_l(Q, k, t)) \wedge w_l(Q, k, t) \wedge v_l(Q, k, t).$$

The last Boolean product is computed bitwise over the bits of $v_l(Q, k, t)$. EQ is a circuit that takes two L-bit words and generates 1 if and only if the two words are equal. An EQ circuit of constant depth and size $O(L)$ can be easily constructed. The preceding expression can be realized by a constant-depth circuit of size $O(s(Q)T(n)L)$. The second operand, if any, can be computed similarly.

2. *Execute operation:* The operation executed at a step is either an addition/subtraction, or a read from the local or the global memory, or a test

based on comparing two numbers. We have already sketched how to realize addition and comparison with constant-depth polynomial-sized circuits. Subtraction is similar to addition. We describe here the circuit needed to simulate a read from the global memory.

Recall that the values stored in the global memory are represented by the triples $(a_g(k,t), v_g(k,t), w_g(k,t))$, where $1 \le k \le n + P(n)T(n)$, in the circuit. Hence, the subcircuit corresponding to a global read operation must identify the triple whose first component is the operand—say, a_0—computed in step 1, and must generate the corresponding L-bit value stored in that triple. This operation can be accomplished by the following Boolean expression:

$$\bigvee_{k=1}^{n+PT} EQ(a_o, a_g(k,t)) \wedge w_g(k,t) \wedge v_g(k,t),$$

where, as before, EQ is the circuit that reports whether two L-bit words are equal. Clearly, the circuit corresponding to a read from the global memory is of constant depth and of $O((n + P(n)T(n))L)$ size.

3. *Update IC values:* The IC value is incremented unless the current instruction is a **go to** statement. Let G_m be the set of indices j such that the jth instruction of the program of Q is a **go to** m **if** $(cond_j)$. Assume that the $(m - 1)$st instruction is not a **go to** statement. The other case can be dealt with similarly. Then,

$$IC(Q, m, t + 1) = IC(Q, m - 1, t) \bigvee_{j \in G_m} (IC(Q, j, t) \wedge cond_j).$$

Clearly, such a function can be realized with a constant-depth circuit of size $O(s(Q))$.

4. *Select result:* As we have seen, there are several subcircuits constructed to execute all the possible operations. The purpose of this stage is to select appropriately the L-bit result, which will be denoted by $res(Q,t)$.

The particular instruction executed by Q at time t specifies the operation; hence, the index j for which $IC(Q,j,t) = 1$ can be used to select the correct result. If the instruction loads a constant or the processor ID, the result is given trivially. Either way, the corresponding circuit is of constant depth and of size $O(s(Q)L)$.

5. *Update memory:* Once the appropriate result is selected, all memory triples needed to simulate step $t + 1$ have to be generated. These triples will be the same as the triples given at the beginning of step t except for those triples whose first component is the address of the location written into at step t. Let $\alpha(Q, t)$ be the L-bit address of the result of the instruction executed by Q at time t. Such a word can be deduced immediately from the instruction executed at this step, unless it is an indirect

write instruction. In the latter case, we use the same type of subcircuit as the one used to compute the operands in step 1.

We almost have all the information needed to set the memory triples. The only item missing is the determination of a local versus global memory change.

Let $\gamma(Q,t)$ be a Boolean function such that $\gamma(Q,t) = 1$ if and only if Q stores the result in the global memory (at time t). We define the function $\lambda(Q,t)$ for a local memory change in a similar fashion. Let $C(Q)$ be the set of those instruction numbers that are global memory write instructions. Then,

$$\gamma(Q,t) = \bigvee_{j \in C(Q)} IC(Q,j,t).$$

The Boolean function corresponding to $\lambda(Q,t)$ is similar. Both functions can be realized with constant-depth circuits of size $O(s(Q))$.

We are ready to specify the local memory triples needed for step $t+1$. First, we set $a_l(Q,k,t+1) = a_l(Q,k,t)$ and $v_l(Q,k,t+1) = v_l(Q,k,t)$, for all $1 \le k \le t$. As for the values $w_l(Q,k,t+1)$, they can be set to $w_l(Q,k,t)$, except for those triples whose first component is equal to $\alpha(Q,t)$, in which case the value of $w_l(Q,k,t+1)$ must be set to 0. We shall use $k = t+1$ to record the new values. That is, we set $a_l(Q,t+1,t+1) = \alpha(Q,t)$, $v_l(Q,t+1,t+1) = res(Q,t)$ and $w_l(Q,t+1,t+1) = \lambda(Q,t)$.

As for specifying the global memory triples, we have to adjust $\gamma(Q,t)$ such that we force the priority-write assumption of the CRCW PRAM. More precisely, $\gamma(Q,t)$ must be set to 0 if there is a lower indexed processor trying to write into the same location. By enforcing the priority-write assumption, we can derive the following Boolean function:

$$\bar{\gamma}(Q,t) = \gamma(Q,t) \wedge \left(\bigvee_{\{S \mid PID(S) < PID(Q)\}} \gamma(S,t) \wedge EQ(\alpha(Q,t), \alpha(S,t)) \right)'.$$

We can now incorporate the new values into the triple $a_g(n + tP + PID(Q), t+1) = \alpha(Q,t)$, $v_g(n + tP(n) + PID(Q), t+1) = res(Q,t)$ and $w_g(n + tP(n) + PID(Q), t+1) = \bar{\gamma}(Q,t)$. It is easy to check that the new memory triples can be implemented with a constant-depth circuit of size $O((n + P(n)T(n))LP(n))$.

The preceding circuitry is replicated $T(n)$ times; hence, the resulting circuit will be of depth $O(T)$ and of size polynomial in $P(n)$, $T(n)$ and n. The output nodes of the circuit are specified from the global memory triples representing the output of the CRCW PRAM. □

We now turn our attention to proving a lower bound on any circuit that computes the parity function.

10.4.2 LOWER BOUNDS FOR UNBOUNDED FAN-IN CIRCUITS

We start by stating an immediate corollary to Theorem 10.10, which will form the basis for establishing lower bounds for the priority CRCW PRAM.

Corollary 10.4: *Suppose we are given a function f such that f can be computed in time $T(n)$ on the restricted priority CRCW PRAM, using a polynomial number of processors. Then, there exists a polynomial-sized circuit of depth $O(T(n))$ that can compute f.* □

This corollary can be used to deduce lower bounds for the priority CRCW PRAM given lower bounds for circuits. Typically, it is less difficult to establish lower bounds for circuits, since they have a static structure. However, we should not conclude that it is easy to obtain lower bounds for circuits. The task is still quite challenging, and known lower bounds require intricate techniques.

We next describe an algebraic approach to derive a lower bound for the circuit computing the parity function.

Approximators. Let $f(x_1, x_2, \ldots, x_n)$ be a Boolean function to be realized by an unbounded fan-in circuit. This circuit can be used to obtain a polynomial $p(x_1, \ldots, x_n)$ of small degree such that p can be viewed as an approximation to the function f. More precisely, we define a **δ-approximator** to be a polynomial $p(x_1, x_2, \ldots, x_n)$ of degree δ over the field $Z_3 = \{-1, 0, 1\}$ such that p takes on the value 0 or 1 on all the inputs from $\{0, 1\}^n$. The **error** introduced by an approximator is the number of input settings from $\{0, 1\}^n$ for which the values of f and p differ.

An approximator p_f **represents** f if no error is introduced by p_f. That is, p_f is a polynomial in the variables $x_1, x_2, \ldots, x_n$ with coefficients from $\{-1, 0, 1\}$ such that $p_f(x_1, x_2, \ldots, x_n) = f(x_1, x_2, \ldots, x_n)$, for all $(x_1, x_2, \ldots, x_n) \in \{0, 1\}^n$. Note that the value of p_f at a particular point is computed over the field Z_3.

EXAMPLE 10.5:

Consider the complement function $f(x) = x'$. Then, $p_f(x) = 1 - x$ represents f, since p_f agrees with f on the inputs $x = 0$ and $x = 1$. More generally, if $f(x_1, \ldots, x_n) = [g(x_1, \ldots, x_n)]'$ and p_g represents g, then $p_f = 1 - p_g$ represents f. □

EXAMPLE 10.6:

The Boolean AND function can be trivially represented by the polynomial $p_{AND} = x_1 \ldots x_n$. In particular, the minterm $m(x_1, \ldots, x_n) = x_{i_1} \ldots x_{i_s} x'_{j_1} \ldots$

x_{j_t}', where $\{i_1, \ldots, i_s\}$ and $\{j_1, \ldots, j_t\}$ are a partition of $\{1, 2, \ldots, n\}$, can be represented by $p_m = x_{i_1} \ldots x_{i_s}(1 - x_{j_1}) \cdots (1 - x_{j_t})$. □

EXAMPLE 10.7:

Consider the Boolean OR function $f(x_1, \ldots, x_n) = x_1 + \cdots + x_n$. Then, using the representations of the complement and the Boolean product functions, we can deduce the representation $p_f = 1 - (1 - x_1) \cdots (1 - x_n)$ of f.

A δ-approximator for f is given by $p = 1 - (1 - x_1) \cdots (1 - x_\delta)$. This approximator disagrees with f on $2^{n-\delta} - 1$ input settings (all input settings $x_1 = x_2 = \cdots = x_\delta = 0$ and at least one $x_j = 1, j > \delta$). □

Lemma 10.7: *Let $f(x_1, \ldots, x_n)$ be any Boolean function. Then, f can be represented by an nth-degree multilinear polynomial over the field $Z_3 = \{-1, 0, 1\}$.*

Proof: Since f can always be written as the sum of minterms, we can use the facts illustrated in Examples 10.6 and 10.7 to obtain a polynomial representation for f. We can make each term of p multilinear using the fact that $x_i^2 = x_i$, for all $1 \leq i \leq n$, on all inputs from $\{0, 1\}^n$. □

Derivation of an Approximator from a Circuit. We are ready to show that a circuit of an arbitrary function f can be used to derive an approximator for f. The degree and error of the approximator will depend on the size and the depth of the circuit.

Lemma 10.8: *Let C be an unbounded fan-in circuit of depth d realizing a function f on n variables. Then, a $(2l)^d$-approximator can be derived from C such that the error introduced by the approximator is at most size$(C)2^{n-l}$, for any integer $l \geq 1$.*

Proof: We will assign inductively an approximator with each subcircuit of C starting with the inputs and working upward level by level to the output gate. Such an assignment will be carried out in such a way that each subcircuit of height h will be assigned a $(2l)^h$-approximator, which may introduce an additional error of at most 2^{n-l}.

Let D be a subcircuit of C of height h. Assume that we have already assigned $(2l)^{h-1}$-approximators to all the subcircuits feeding directly into the top gate of D. The function assigned to D will depend on the top gate.

Let us begin with the easy case, where the top gate is a NOT gate. Let the circuit feeding into it be assigned the approximator g. Then, assign the polynomial $1 - g$ to the subcircuit D. Clearly, $1 - g$ is a $(2l)^h$-approximator, which introduces no additional error.

Assume that the top gate is an OR gate. We can reduce the case of an AND gate to the OR case by using NOT gates.

Let $f_1, f_2, \ldots, f_k$ be the $(2l)^{h-1}$-approximators assigned to the subcircuits feeding into the output OR gate of D. Consider a set of l polynomials $g_i = \left(\sum_{j=1}^{k} \alpha_{ij} f_j\right)^2$ over Z_3, where $\alpha_{ij} \in \{0, 1\}$ and $1 \leq i \leq l$. Notice that the values of the g_i's are always 0 or 1 for all inputs over Z_3. Assign the approximator $f_D = 1 - (1 - g_1)(1 - g_2) \cdots (1 - g_l)$ to the subcircuit D. The polynomial f_D is clearly a $(2l)^h$-approximator, since the degree of f_D is $\leq (2l)^h$ and the output of f_D is 0 or 1 for all inputs from $\{0, 1\}^n$. It remains to obtain an estimate on the error introduced by f_D.

Consider any input setting I for which the approximators $f_1, f_2, \ldots, f_k$ generate the correct values. If the ouput of the top gate of D is 0, then we must have $f_1(I) = f_2(I) = \cdots = f_k(I) = 0$. This result implies that $g_1(I) = g_2(I) = \cdots = g_l(I) = 0$, regardless of the α_{ij}'s, and hence $f_D(I) = 0$ and no error is introduced in this case. Assume now that the output of the top gate of D is 1. Then, there exists an index j_0 such that $f_{j_0}(I) = 1$. Given any set of values α_{ij}, where $j \neq j_0$, there exists a *unique* set of l values $\alpha_{1j_0}, \alpha_{2j_0}, \ldots, \alpha_{lj_0}$ that will force each g_i to be equal to 1—that is, will force $f_D = 0$. Therefore, if we pick all the $\alpha_{ij} \in \{0, 1\}$ at random and independently of one another, the probability that $f_D = 0$ on such an input is at most 2^{-l}. This fact implies that the expected number of input values for which $f_D = 0$ is at most 2^{n-l}. Hence, there exists a set of values of the α_{ij}'s for which the corresponding $(2l)^h$-approximator introduces an error on at most 2^{n-l} input settings.

The proof of the lemma can be concluded as follows. Assign 1-approximators to the inputs of the circuit and perform the general assignment described previously, level by level. The output will be assigned a $(2l)^d$-approximator with at most $size(C)2^{n-l}$ error, since each approximator of a subcircuit introduces at most 2^{n-l} error. □

Setting $l = \frac{1}{2}n^{\frac{1}{2d}}$, we obtain the following corollary.

Corollary 10.5: *Any circuit C of depth d has a $\sqrt{n}$-approximator, which introduces an error on at most $size(C)2^{n-\frac{1}{2}n^{\frac{1}{2d}}}$ input settings.* □

We now concentrate on the parity function.

Approximators for the Parity Function. Let $\pi(x_1, x_2, \ldots, x_n)$ be the parity function on n variables. Corollary 10.5 implies that any circuit of depth d computing π can be used to deduce a $\sqrt{n}$-approximator that differs from π on at most $size(C)2^{n-\frac{1}{2}n^{\frac{1}{2d}}}$. We show next that *any* $\sqrt{n}$-approximator of π will have to disagree with π on at least $\frac{1}{50}2^n$ input settings. This fact will be used to deduce a lower bound on $size(C)$ as a function of d.

Lemma 10.9: *Let $a(x_1, x_2, \cdots, x_n)$ be a $\sqrt{n}$-approximator for the parity function $\pi(x_1, x_2, \cdots, x_n)$. Then, a and π have to disagree on at least $\frac{1}{50}2^n$ input settings for sufficiently large n.*

Proof: Recall that $a(x_1, x_2, \cdots, x_n)$ is a polynomial over $Z_3 = \{-1, 0, 1\}$ such that a maps $\{0, 1\}^n$ into $\{0, 1\}$. We make the change of variables $y_i = x_i + 1$ for each i. Note that, for $x_i = 0$, we have $y_i = 1$; for $x_i = 1$, we have $y_i = -1$. With this change of variables, $a(y_1, y_2, \cdots, y_n)$ maps $\{-1, 1\}^n$ into $\{-1, 1\}$. In addition, the parity function $\pi(y_1, y_2, \cdots, y_n)$ is now a mapping from $\{-1, 1\}^n$ into $\{-1, 1\}$, which can be represented by the polynomial $p_\pi(y_1, y_2, \cdots, y_n) = y_1 y_2 \cdots y_n$.

Let $G \subset \{-1, 1\}^n$ be the set of inputs on which $a(y_1, \cdots, y_n)$ and $p_\pi(y_1, \cdots, y_n)$ agree. We will continue with the proof of the lemma after we look at the following Claim and its Proof.

Claim: Let f be any function from G into $\{-1, 0, 1\}$. Then, using the approximator $a(y_1, \cdots, y_n)$, f can be represented by a multilinear polynomial $p_f(y_1, y_2, \cdots, y_n)$ whose degree is at most $\frac{n}{2} + \sqrt{n}$.

Proof: We extend f arbitrarily to $\bar{f} : \{-1, 1\}^n \to \{-1, 0, 1\}$. Let $p_{\bar{f}}$ be a polynomial representing $\bar{f}$. Then $p_{\bar{f}}$ can be expressed as a sum of terms such that each term is multilinear given that $y^2 = 1$ for any variable y. Let $cy_{i_1}y_{i_2} \cdots y_{i_s}$ be a term of $p_{\bar{f}}$ such that $c \neq 0$. If $s > \frac{n}{2}$, we will show how to replace this term with another term of degree at most $\sqrt{n} + n - s$, such that both terms take on identical values on inputs from G. Let $\{j_1, j_2, \cdots, j_t\} = \{1, 2, \cdots, n\} - \{i_1, i_2, \cdots, i_s\}$. We replace $cy_{i_1}y_{i_2} \cdots y_{i_s}$ with the equivalent expression $cy_1y_2 \cdots y_n y_{j_1}y_{j_2} \cdots y_{j_t}$, again using the fact that $y^2 = 1$. But $y_1y_2 \cdots y_n$ can be replaced with a, since a is an approximator of $p_\pi = y_1 \cdots y_n$, and they both agree on G. The degree of the new term is clearly $\sqrt{n} + n - s$. Making such substitutions for all terms of degree greater than $\frac{n}{2}$ will result in a polynomial representing f of degree at most $\frac{n}{2} + \sqrt{n}$. This result concludes the proof of the claim.

We now continue with the proof of Lemma 10.9. We have shown that each function $f : G \to \{-1, 0, 1\}$ can be represented by a multilinear polynomial of degree $\frac{n}{2} + \sqrt{n}$. The number of such functions is $3^{|G|}$. On the other hand, the number of multilinear terms of degree $\leq \frac{n}{2} + \sqrt{n}$ is given by

$$\sum_{i=0}^{\frac{n}{2}+\sqrt{n}} \binom{n}{i} \approx .9772 \times 2^n < \frac{49}{50}2^n.$$

This implies that the number of all polynomials with these types of terms over Z_3 is $< 3^{\frac{49}{50}2^n}$ since each such polynomial is of the form $\sum_i \beta_i m_i$, $\beta_i \in Z_3$ and m_i is a multilinear term. Therefore, $3^{|G|} < 3^{\frac{49}{50}2^n}$, which implies that $|G| < \frac{49}{50}2^n$. We conclude that the number of input settings for which a and π have

to disagree is at least $2^n - \frac{49}{50}2^n = \frac{1}{50}2^n$. The proof of the lemma is now complete. □

Lower Bounds on the Parity Function. We begin by deducing a lower bound on the size of any circuit computing the parity function in depth d.

Theorem 10.11: *Let C be any circuit of depth d that computes the parity function $\pi(x_1, x_2, \ldots, x_n)$. Then, the size of C must satisfy $size(C) \geq \frac{1}{50}2^{\frac{1}{2}n^{\frac{1}{2d}}}$.*

Proof: The circuit C can be used to obtain a $\sqrt{n}$-approximator for π that introduces at most $size(C)2^{n-\frac{1}{2}n^{\frac{1}{2d}}}$ error. On the other hand, by Lemma 10.9, this error must be at least $\frac{1}{50}2^n$. Therefore $size(C)2^{n-\frac{1}{2}n^{\frac{1}{2d}}} \geq \frac{1}{50}2^n$, from which the theorem follows. □

Corollary 10.6: *Any polynomial size circuit that computes the parity function on n variables must have depth at least $\Omega\left(\frac{\log n}{\log \log n}\right)$.*

Proof: Suppose that C is a circuit that computes the parity function such that $size(C) \leq n^k$, for some constant k. Then, $n^k \leq \frac{1}{50}2^{\frac{1}{2}n^{\frac{1}{2d}}}$, from which the lower bound on d follows. □

We are now ready to state our main lower bound on the priority CRCW PRAM.

Corollary 10.7: *A restricted priority CRCW PRAM computing the parity function of n variables with a polynomial number of processors requires at least $\Omega\left(\frac{\log n}{\log \log n}\right)$ time.*

Proof: Suppose that we are given a CRCW PRAM algorithm that computes the parity function of n variables in time $T(n)$ using a polynomial number of processors $P(n)$. By Theorem 10.10, we can construct a circuit C that computes the parity function such that the depth of C is $O(T(n))$ and the size of C is polynomial in n (since $P(n)$ is polynomial in n). Using Corollary 10.6, we obtain that $T(n) = \Omega\left(\frac{\log n}{\log \log n}\right)$. □

The lower bound on the time it takes to compute the parity function holds even on the ideal priority CRCW. The corresponding proof, however, is substantially more involved; refer to the bibliographic notes for references.

We can use our lower bound on computing the parity function to deduce a lower bound on the majority function, as shown by the next lemma.

Lemma 10.10: *The majority function, defined by* $f_m(x_1, x_2, \ldots, x_n) = 1$ *if and only if the number of* 1*'s among* $x_1, x_2, \ldots, x_n$ *is* $\geq \frac{n}{2}$, *requires* $\Omega\left(\frac{\log n}{\log \log n}\right)$ *time on the restricted priority CRCW PRAM, using a polynomial number of processors.*

Proof: We shall use a reduction from the parity function to the majority function. That is, we shall provide an efficient way to compute the parity function by using an algorithm to compute the majority function.

Let l be any integer between 1 and n. The **threshold** function is defined by $th_l(x_1, x_2, \ldots, x_n) = 1$ if and only if there are at least l 1's among $x_1, \ldots, x_n$.

Let $z = (\underbrace{00\cdots0}_{l}\ \underbrace{11\cdots1}_{n-l})$ consisting of l consecutive 0's followed by $n - l$ consecutive 1's. The majority function applied to the input (x, z) generates 1 if and only if $th_l(x_1, x_2, \ldots, x_n) = 1$. Hence, for each index l, we can use an algorithm for the majority function to compute $th_l(x_1, x_2, \ldots, x_n)$. In particular, n parallel applications of the majority algorithm will generate $th_l(x_1, x_2, \ldots, x_n)$, for all $1 \leq l \leq n$.

Let $e_l(x_1, x_2, \ldots, x_n) = th_l(x_1, x_2, \ldots, x_n) \wedge (th_{l+1}(x_1, x_2, \ldots, x_n))'$, for all odd indices $1 \leq l \leq n$, assuming, without loss of generality, that n is even. It is clear that, once all the threshold functions are generated, the functions e_l's can be obtained in $O(1)$ parallel steps, using n processors. In particular, the parity function can be expressed as $\pi = e_1 \vee e_3 \vee e_5 \vee \ldots \vee e_{n-1}$. Therefore, we can compute the parity function, by using n applications of the majority algorithm, plus $O(1)$ steps, using n processors. It follows that, with a polynomial number of processors, the running time of the corresponding parity-function algorithm is bounded by the running time of the majority algorithm, and thus the corollary follows. □

The parity function can be reduced to several other important problems. Some of these reductions are left to Exercises 10.20 and 10.21.

10.5 Introduction to *P*-Completeness

We have described throughout this book many techniques to design efficient parallel algorithms for a large variety of problems. Our implicit goal has been to develop PRAM algorithms that are very fast; that is, the algorithms run in $O(\log^k n)$ time for some small fixed constant k, and use a total number of operations comparable to the time complexity of the best known sequential algorithm. Recall from Section 1.5 that the performance of such algorithms can be expressed in the time-processors framework as follows.

Let $T(n)$ and $W(n)$ be, respectively, the running time and the total number of operations of the parallel algorithm. With p processors, the parallel algorithm runs in $O\left(\frac{W(n)}{p} + T(n)\right)$ time (by Brent's scheduling principle). Therefore, the parallel algorithm runs roughly p times faster than the best known sequential algorithm, as long as p is not too large. Problems solvable by this type of algorithms are clearly ideal for parallel processing.

We are not always so lucky. Other problems seem to exhibit different degrees of "parallelizability" as a function of the associated cost.

10.5.1 WHEN IS A PROBLEM PARALLELIZABLE?

Unlike the problems discussed so far in this book, many other problems appear to be much less amenable to parallel processing, even with the presence of a relatively large number of processors. An interesting avenue to take would be to classify problems according to a certain well-defined notion of **parallelizability.** Such a notion is complicated by the fact that we have to deal with at least two parameters simultaneously—time and number of processors. Let us consider the following two extremes.

1. A problem can be defined to be parallelizable if it can be solved *faster as we increase the number of processors* within a certain range—say, $1 \leq p \leq \alpha(n)$—where $\alpha(n)$ is an increasing function of n. Optimality is achieved if the parallel algorithm is $\Theta(p)$ times faster than the best known sequential algorithm within the given range of p.

   For the optimal algorithms described in this book, the range of p is given by $1 \leq p \leq \frac{n}{\log^k n}$, for some fixed constant k. This intuitive notion is appealing as long as the range of p is large. However, when $\alpha(n)$ is small, it becomes less clear why such a problem should be viewed as parallelizable, since no gain can be achieved beyond $p = \alpha(n)$.

2. A problem can be defined to be parallelizable if it can be solved *extremely fast with a reasonable number of processors*. An algorithm will be considered to be extremely fast if its running time is *polylogarithmic*—that is, $O(\log^k n)$ for some fixed constant k. We now have to define what we mean by a reasonable number of processors. We can borrow our definition from circuit complexity, where the circuit model itself can be used as a model for parallel computation. In the circuit model, a reasonable amount of hardware is used when the size of the circuit is polynomial in the input length. Therefore, we can define a parallelizable problem to be a problem that can be solved in polylogarithmic time, using a polynomial number of processors. Clearly, this notion captures the desire to obtain dramatic speedups at a certain cost.

The complexity class *NC*, to be introduced more formally later, consists of all the problems that can be solved by polylogarithmic-time algorithms on a PRAM, using a polynomial number of processors. Given our earlier simulation results (Section 10.1), the class *NC* remains invariant under our various PRAM models. It follows that a problem not in *NC* cannot be solved extremely quickly with a polynomial number of processors.

As the following example shows, however, the fact that a problem belongs to the class *NC* does not necessarily imply that some speedup is possible, given a certain number of processors.

EXAMPLE 10.8:

We can solve the problem of identifying a breadth-first search tree of an undirected graph $G = (V, E)$ by using the shortest-paths algorithm described in Section 5.5. Let $|V| = n$ and $|E| = m$. Then, the corresponding parallel algorithm runs in $O(\log^2 n)$ time, using $O(M(n) \log n)$ operations, where $M(n)$ is the sequential complexity of $n \times n$ matrix-multiplication. Therefore, this problem is in *NC*. Let us now express the complexity bounds of the parallel algorithm in the time-processors framework. If we use the standard (and practical) matrix-multiplication algorithm, the parallel algorithm will run in $O\left(\frac{n^3 \log n}{p} + \log^2 n\right)$ time with p processors. The best known sequential algorithm, which runs in $O(m + n)$ time, is *faster* than the parallel algorithm for all values of p satisfying $p = o(n \log n)$ even if the graph is dense. □

Example 10.8 suggests that we have to be careful in interpreting the fact that a problem is in *NC*. Our intention in this section is to provide strong evidence that some of the important problems are *not* in *NC*, implying that no dramatic speedups are possible with any polynomial number of processors.

We next introduce three of these problems.

10.5.2 ORDERED DEPTH-FIRST SEARCH

Let $G = (V, E)$ be a directed graph represented by its adjacency lists; that is, we have a list $Adj(v)$ containing all the nodes u such that $(v, u) \in E$, for each vertex v. Given a *starting vertex* s, we perform a depth-first traversal of G by visiting all the nodes of G, using the following procedure. Begin by visiting s. Let v be an arbitrary vertex. When visiting v for the first time, we mark v as visited. Next, we pick the first unvisited vertex u on the list $Adj(v)$, if such a vertex exists, and we then apply the search procedure at u. If no such vertex exists the search procedure at v terminates, and the search resumes at the last vertex visited before v. Each vertex v is assigned a number $dfs(v)$ indicating its

order in the depth-first search of G. Figure 10.6 shows a graph with the corresponding numbering.

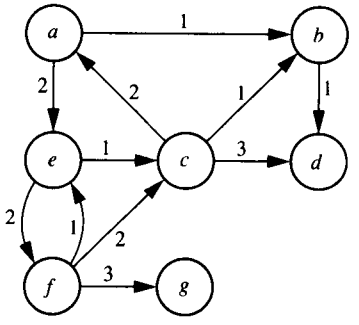
Depth-first search is an important technique for designing efficient sequential algorithms on graphs. It can be implemented easily in linear time. Later, we shall show that ordered depth-first search is probably outside the class NC, and hence it is unlikely that ordered depth-first search can be solved by a polylogarithmic-time algorithm, using a polynomial number of processors. However, optimality can be achieved when the number of processors belongs to a certain range, as stated in Exercise 10.22.

10.5.3 MAXIMUM FLOW

A **network** is a directed graph $N = (V, A)$ with two distinguished vertices, the **source** s and the **sink** t, such that each arc e is labeled with a positive integer $c(e)$, called the **capacity** of e. A **flow** f in a network N is an integer-valued function f defined on the set of arcs such that

1. $0 \leq f(e) \leq c(e)$, for all arcs e (**capacity constraint**).
2. $\sum_u f(u, v) = \sum_w f(v, w)$, for every node $v \in V - \{s, t\}$ (**conservation constraint**).

This type of network can be used to model transportation problems in which commodities are shipped from a source to a destination. The capacity is then the maximum rate at which commodity can be transported. The first



(a)

Vertices	dfs Numbers
a	1
b	2
c	5
d	3
e	4
f	6
g	7

(b)

FIGURE 10.6
Graph for ordered depth-first search. (a) A directed graph with the arcs numbered according to their order of appearance on the adjacency lists. (b) The depth-first search numbers of the vertices, with a as the starting vertex.

constraint states that no commodity flow can exceed the capacity; the second constraint states that the sum of the flow on all incoming arcs to an arbitrary node $v \neq s, t$ is equal to the sum of the flow on all outgoing arcs.

Let $f^+(v) = \sum_w f(v, w)$ and $f^-(v) = \sum_u f(u, v)$. The **value** of a flow f is equal to $val(f) = f^+(s) - f^-(s)$. Because of the conservation constraint, the value of the flow has to appear at the sink and hence $val(f) = f^-(t) - f^+(t)$. A flow is **maximum** if $val(f) \geq val(f')$ for any other flow f' in N. Figure 10.7 shows an example of a network and a corresponding maximum flow.

The problem of network flows has been studied extensively in the past 40 years. We note here only that several efficient algorithms with running

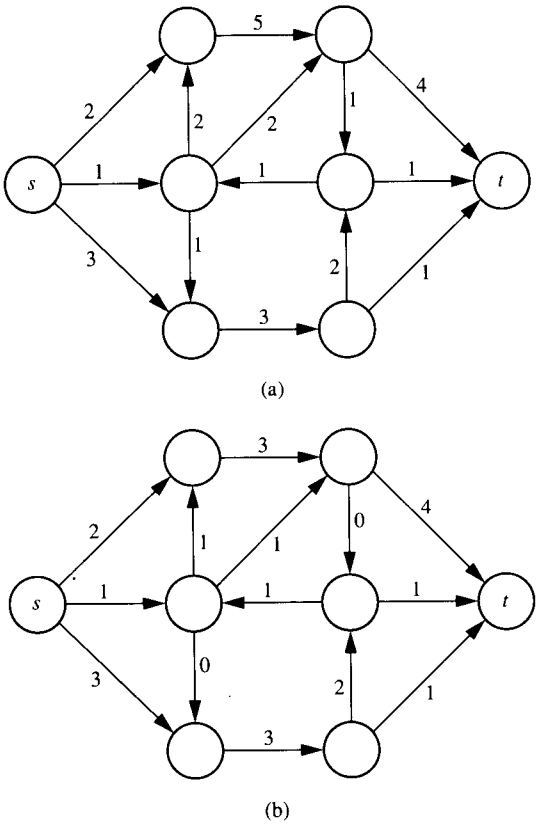


FIGURE 10.7
Network and corresponding maximum flow. (a) A network with the edge capacities. (b) A maximum flow of value 6.

time $O(n^3)$ or faster are known (see bibliographic notes). As for the parallel complexity of this problem, there are no known polylogarithmic-time algorithms using a polynomial number of processors that solve the general network-flow problem. Again, we shall provide strong evidence that no such parallel algorithms exist.

10.5.4 LINEAR PROGRAMMING

We encountered the special version of linear programming with two variables when we studied intersections of convex sets (Section 6.2). In general, we are supposed to optimize a linear function subject to a set of linear constraints. One possible formulation is given next:

Minimize $\sum_i^n c_i x_i$
Subject to
$Ax \leq b$
$x_i \geq 0, 1 \leq i \leq n$

Linear programming is of fundamental importance in both theory and practice. Polynomial-time algorithms have been discovered recently that solve the general linear-programming problem. However, as in the previous two examples, linear programming is likely to lie outside the class *NC*, and hence dramatic speed-ups are unlikely if we use a polynomial number of processors.

We are now ready to introduce more formally the class *NC* and its randomized version *RNC*.

10.5.5 THE CLASSES *NC* AND *RNC*

Complexity classes are typically defined in terms of languages. A **language** L is simply a subset of strings over the alphabet $\{0, 1\}$. An algorithm to recognize a language L takes an arbitrary string $x \in \{0, 1\}^*$ and determines whether or not x is in L. Clearly, the language-recognition problem is a **decision problem** whose answer is either yes or no.

We would like to deal with language-recognition problems, since the theory is simpler and the details omitted are superfluous to the theory. We can easily associate decision problems with any computational problem. For example, we can convert the maximum-flow problem into a decision problem by asking whether the ith bit of the maximum flow value is 1 or 0. In general, a computational problem is at least as difficult to solve as is any decision

problem associated with it. Hence, evidence indicating that a decision problem is not parallelizable can be used to show that the computational problem itself is not parallelizable.

The **class *NC*** is defined to be the set of all languages L such that, on all inputs of length n, L can be recognized by a PRAM algorithm running in $O(\log^k n)$ time, using a polynomial number of processors, where k is a constant that does not depend on n. Similarly, the **class *RNC*** is the set of all languages that can be recognized by a **randomized** PRAM algorithm in polylogarithmic time using a polynomial number of processors. As we saw in Chapter 9, there are two general types of randomized algorithms, Las Vegas and Monte Carlo, each of which gives rises to its own randomized *NC* class.

All the problems discussed in the previous chapters belong to the class *NC*, except for the *maximum-matching problem*, which was shown to be in *RNC*. In what follows, we restrict our attention to the class *NC*. The notions introduced for the class *NC* can also be adapted to the class *RNC*.

The Classes *P* and *NC*. Let P be the class of all languages that can be recognized by a deterministic Turing machine within a polynomial number of steps. The class P remains invariant under various models of sequential computations, and, in particular, P can be defined as the class of all problems that can be solved on a RAM in polynomial time. The class P is universally accepted to represent the class of all the problems that can be solved efficiently on a sequential processor.

There is an intriguing relationship between the classes P and *NC*. On one hand, we can see that $NC \subseteq P$ by converting *NC*-algorithms into sequential algorithms in the obvious way. A fundamental problem in complexity theory is whether or not $P \subseteq NC$. The popular wisdom seems to suggest that $P \not\subseteq NC$, and hence there are problems in P that cannot be solved quickly in parallel with a polynomial number of processors. The class of P-complete problems, to be introduced shortly, consists of the most likely candidates of P that are not in *NC*. As in the case of *NP*-complete problems, the notion of reducibility is of utmost importance.

The Reducibility Notion. Let L_1 and L_2 be two languages. The language L_1 is ***NC*-reducible** to the language L_2 if there exists an *NC*-algorithm that transforms an arbitrary input u_1 for L_1 into an input u_2 for L_2 such that $u_1 \in L_1$ if and only if $u_2 \in L_2$.

Note that the notion of reducibility is not symmetric, and that an *arbitrary* instance of L_1 is transformed into a potentially special instance of L_2. The existence of the *NC* transformation implies an algorithm for recognizing L_1 given any algorithm for recognizing L_2. In particular, we have the following lemma.

Lemma 10.11: *Let L_1 and L_2 be two languages such that L_1 is NC-reducible to L_2. Then, $L_2 \in NC \rightarrow L_1 \in NC$.*

Proof: Since L_1 is *NC*-reducible to L_2, there exists an *NC*-algorithm A that transforms an arbitrary input u_1 for L_1 into an input u_2 for L_2 such that $u_1 \in L_1$ if and only if $u_2 \in L_2$. Let u_1 be an arbitrary input of L_1. To determine whether $u_1 \in L_1$, we use the following procedure. Apply A to u_1, and let u_2 be the resulting input for L_2. Since $L_2 \in NC$, apply the corresponding *NC*-algorithm to determine whether $u_2 \in L_2$. We can now immediately decide whether or not $u_1 \in L_1$, since $u_1 \in L_1$ if and only if $u_2 \in L_2$. The resulting procedure is clearly in *NC*. □.

Another important fact about *NC*-reducibility, the proof of which is left to Exercise 10.26, is stated in the next lemma.

Lemma 10.12: *If L_1 is NC-reducible to L_2, and L_2 is NC-reducible to L_3, then L_1 is NC-reducible to L_3; that is, NC-reducibility is transitive.* □

The Notion of *P*-Completeness. We now come to the central notion of *P*-completeness. A language L is ***P*-complete** if (1) $L \in P$, and (2) every language in P is *NC*-reducible to L. From this definition, we can immediately deduce the following fact.

Lemma 10.13: *Let L be a P-complete problem. If $L \in NC$, then $NC = P$.* □

Lemma 10.13 can be rephrased as follows. If $P \neq NC$, as is widely conjectured, then no *P*-complete problem belongs to *NC*. Therefore, a *P*-complete problem represents a likely candidate of P that cannot be solved very fast on the PRAM with a polynomial number of processors.

Up to this point, however, it is not clear whether *P*-complete problems exist. Next, we introduce the circuit value problem, and show that it belongs to the class of *P*-complete problems.

The Circuit Value Problem (CVP). The **circuit value problem (CVP)** is to determine the value of the single output of a Boolean circuit consisting of NOT gates and two-valued AND and OR gates, for a given set of inputs. More precisely, our circuit C is specified by a sequence $C = \langle g_1, g_2, \ldots, g_n \rangle$, where each g_i is either an input or $g_i = g_j \vee g_k$, or $g_i = g_j \wedge g_k$, or $g_i = \neg g_j$, and $j, k < i$.

We state the CVP as follows: Given a Boolean circuit C represented by a sequence $C = \langle g_1, g_2, \ldots, g_n \rangle$, and a specified set of inputs, determine whether the value of the circuit is equal to 1.

We will show next that CVP is *P*-complete. Since this is our first *P*-complete problem, the proof has to show that an arbitrary language $L \in P$ is *NC*-reducible to CVP.

Theorem 10.12: *CVP is P-complete.*

Proof: Given the sequence $C = \langle g_1, g_2, \ldots, g_n \rangle$ and a set of input values, the values computed at the gates g_i's can be determined consecutively, starting with g_1 and proceeding with $g_2, g_3, \ldots, g_n$. Hence, the output value can be determined in $O(n)$ sequential time, which implies that CVP is in *P*.

We show next that, given an arbitrary $L \in P$, L is *NC*-reducible to CVP. That is, we provide an *NC*-algorithm that takes an arbitrary instance I_L of the language L and maps I_L into an instance I_{CVP} of CVP such that I_L has a yes answer if and only if I_{CVP} has a value 1.

Since $L \in P$, there exists a one-tape deterministic Turing machine M that accepts L in polynomial time $T(n)$, whenever the input is of size n. Without loss of generality, we can assume that the input bits appear in n consecutive tape cells numbered 1 to n, and all other cells are initially blank. The tape head is initially scanning cell 1 and, after $T(n)$ steps, the output is written into cell 1. Let $Q = \{q_1, \ldots, q_s\}$ be the set of states of M such that q_1 is the initial state, and let $\Sigma = \{a_1, \ldots, a_m\}$ be the tape alphabet. The state-transition function is given by $\delta : Q \times \Sigma \rightarrow Q \times \Sigma \times \{L, R\}$. We shall show how to construct a Boolean circuit C such that the value of C is 1 if and only if M accepts L (that is, cell 1 contains 1 at time $T(n)$).

The circuit C will be defined by the following Boolean functions:

1. $H(i, t)$, such that $H(i, t) = 1$ if and only if the head scans cell i at time t, for $1 \leq i \leq T(n)$ and $0 \leq t \leq T(n)$.
2. $C(i, j, t)$, such that $C(i, j, t) = 1$ if and only if cell i contains character a_j at time t, for $1 \leq i \leq T(n)$, $1 \leq j \leq s$ and $0 \leq t \leq T(n)$.
3. $S(k, t)$, such that $S(k, t) = 1$ if and only if the state of M is q_k at time t, $1 \leq k \leq s$ and $0 \leq t \leq T(n)$.

The circuit will consist of $T(n)$ levels, such that level l contains the gates computing the functions $H(i, l)$, $C(i, j, l)$, and $S(k, l)$, for all $1 \leq i \leq T(n)$, $1 \leq j \leq m$ and $1 \leq k \leq s$. Hence, the number of gates at each level is $O(T(n))$, since s and m depend on only the Turing machine M. We start by specifying $H(i, 0)$, $C(i, j, 0)$ and $S(k, 0)$, which define the inputs to our circuit. We then show how to express $H(i, t + 1)$, $C(i, j, t + 1)$ and $S(k, t + 1)$ in terms of $H(*, t)$, $C(*, *, t)$ and $S(*, t)$, using the specification of the Turing machine M. These expressions will give us a proper description of our circuit in the required format.

For $t = 0$, the tape head is scanning cell 1, the input bits are stored in cells 1 up to n, and M is in state q_1. The Boolean functions $H(i, 0)$, $C(i, j, 0)$ and $S(k, 0)$ encode these facts and provide the inputs to our circuit C as follows:

$$H(i, 0) = \begin{cases} 1 \text{ if } i = 1; \\ 0 \text{ otherwise.} \end{cases}$$

$$\mathrm{C(i, j, 0)} = \begin{cases} 1 \text{ if cell } i \text{ contains initially } a_j; \\ 0 \text{ otherwise.} \end{cases}$$

$$\mathrm{S(k, 0)} = \begin{cases} 1 \text{ if } k = 1; \\ 0 \text{ otherwise.} \end{cases}$$

We now specify each of $H(i, t + 1)$, $C(i, j, t + 1)$ and $S(k, t + 1)$ separately.

Let $I_R = \{(k, j) \mid \delta(q_k, a_j) = (*, *, R)\}$ and $I_L = \{(k, j) \mid \delta(q_k, a_j) = (*, *, L)\}$. That is, I_R consists of all pairs (q_k, a_j) that cause M to move its head right, and I_L consists of all pairs that cause M to move its head left. Now, $H(i, t + 1)$ can be defined as follows:

$$H(i, t + 1) = H(i - 1, t) \sum_{(k,j) \in I_R} C(i - 1, j, t) S(k, t) + H(i + 1, t) \sum_{(k,j) \in I_L} C(i + 1, j, t) S(k, t),$$

where $+$ and Σ indicate Boolean sums, and juxtaposition indicates Boolean product.

It is clear that $H(i, t + 1)$ can be expressed by a sequence of AND and OR gates of length $O(|Q||\Sigma| = sm)$, which is constant. Clearly, all $H(i, t)$'s can be generated in $O(1)$ time, using $O(T^2(n))$ processors.

We next consider $C(i, j, t + 1)$. Recall that $C(i, j, t + 1)$ represents the fact that cell i contains a_j at time $t + 1$. Hence, $C(i, j, t + 1) = 1$ if either cell i contains a_j and the tape head does not scan cell i at time t, or the tape head writes a_j in cell i at time $t + 1$. Hence,

$$C(i, j, t + 1) = \overline{H(i, t)} C(i, j, t) + H(i, t) \sum_{\{(k,j') \mid \delta(q_k, a_{j'}) = (*, a_j, *)\}} C(i, j', t) S(k, t).$$

As in the case of $H(i, t + 1)$, the function $C(i, j, t + 1)$ can be expressed by a constant-length sequence of AND and OR gates. And all the functions $C(i, j, t)$'s can be generated in $O(1)$ time, using $O(T^2(n))$ processors.

We finally consider $S(k, t + 1)$. Let $I_k = \{(k', j) \mid \delta(q_{k'}, a_j) = (q_k, *, *)\}$. Then, $S(k, t + 1)$ can be expressed as follows:

$$S(k, t + 1) = \sum_{1 \le i \le T(n), (k', j) \in I_k} S(k', t) \wedge H(i, t) \wedge C(i, j, t).$$

Hence, the function $S(k, t + 1)$ can be expressed by a sequence of AND and OR gates of length $O(T(n))$. Such a sequence can be generated in $O(\log n)$ time, using $O(T^2(n))$ processors.

The value of the circuit will be given by $C(1, *, T(n))$. From the preceding description, it is clear that the sequence defining the circuit C can be generated by an NC-algorithm, since $T(n)$ is a polynomial in n. The output of C is 1 if and only if M recognizes L. Since L is an arbitrary language in P, it follows that CVP is P-complete. □

As we shall see in Section 10.5.6, CVP and several of its variations provide a powerful tool for establishing the P-completeness of many other problems. We shall look quickly at several variations that are needed for Section 10.5.6.

- **Monotone circuit value problem (MCVP):** Given a Boolean circuit constructed of AND and OR gates only, and a specified set of inputs and their complements, determine whether the value of the circuit is 1.
- **NOR circuit value problem (NOR-CVP):** Given a Boolean circuit $C = \langle g_1, \ldots, g_n \rangle$ such that g_i is either an input equal to 1 or $g_i = \neg(g_j \vee g_k)$ for some $j, k < i$, determine whether the output of C is equal to 1.
- **Fan-out-2 monotone circuit value problem (MCVP2):** We are given a monotone Boolean circuit $C = \langle g_1, \ldots, g_n \rangle$ using AND and OR gates only such that the fan-out of each internal gate is at most 2, the fan-out of each input is at most 1, and g_n is an output OR gate. We are also given a specified set of inputs and their complements. We then wish to determine whether the value of the output is equal to 1.

Theorem 10.13: *The problems MCVP, NOR-CVP, and MCVP2 are all P-complete.*

Proof: We prove only that MCVP is P-complete. The simple proofs for NOR-CVP and MCVP2 are left to Exercises 10.27 and 10.28.

Clearly, MCVP is in P. In our proof of Theorem 10.12, the only place we used negation was in deriving the expression for $C(i, j, t + 1)$. Clearly, $\overline{H(i,t)} = \sum_{1 \le l \le T(n), l \ne l} H(l, t)$, since the tape head has to scan a cell at time t. Otherwise, the proof holds as before, and therefore MCVP is P-complete. □

10.5.6 A SAMPLE OF *P*-COMPLETE PROBLEMS

Now that we have shown that CVP and several of its variations are P-complete, the task of establishing the P-completeness of additional problems is made easier by the following fact.

Lemma 10.14: *Let L be a language that is known to be P-complete. If L is NC-reducible to another language $L' \in P$, then L' is also P-complete.* □

We consider the problems of ordered depth-first search, maximum flow, and linear programming discussed earlier. We associate with them the following decision problems.

- **Ordered depth-first search (ordered DFS):** Given a directed graph $G = (V, E)$ specified by fixed adjacency lists and three vertices s, u and v, determine whether vertex u is visited before vertex v in the depth-first traversal of G starting at s.
- **Maximum flow (MAXFLOW):** Given a network with integer-valued capacities and two distinguished vertices, the source s and the sink t, determine whether the value of the maximum flow is odd.
- **Linear inequalities (LI):** Given an $n \times m$ matrix, and an n-dimensional vector b, all entries being integers, determine whether there exists a rational vector x such that $Ax \leq b$.

Our next goal is to show that ordered DFS, MAXFLOW, and LI are P-complete.

Ordered Depth-First Search.

Theorem 10.14: *Ordered DFS is P-complete.*

Proof: We shall show an NC-reduction from the P-complete problem NOR-CVP to ordered DFS. Since we already know that ordered DFS is in P, such a reduction will establish the theorem.

Let $C = \langle g_1, g_2, \ldots, g_n \rangle$ be an arbitrary instance of the NOR-CVP problem. We shall construct a directed graph G with three special vertices s, u, and v, such that the output of C is 1 if and only if $dfs(u) < dfs(v)$ for an ordered DFS of G starting at s.

The graph G consists of a chain of n components G_i such that G_i is associated with node g_i of the circuit C. A component G_i will share some vertices with other components. There will be two possible ways to traverse a component G_i in a depth-first traversal of G corresponding to whether the value of g_i is 1 or 0. In other words, the value of g_i will uniquely determine the manner in which G_i is traversed, and, conversely, a traversal of G_i will uniquely determine the value of g_i.

We start with the input nodes—that is, all the g_i's whose initial values are equal to 1 (recall that all inputs are equal to 1 in an instance of NOR-CVP). Let g_i be such a node, and let $g_{j_1}, g_{j_2}, \ldots, g_{j_k}$ be the nodes of C in which g_i appears as input. The component G_i is shown in Fig. 10.8. The number next to an edge indicates the edge's rank in the adjacency list of the corresponding vertex. No number is indicated whenever there is only one edge emanating from a vertex. Note that, if $i = 1$, then the node labeled by $exit(i - 1)$ does not exist.

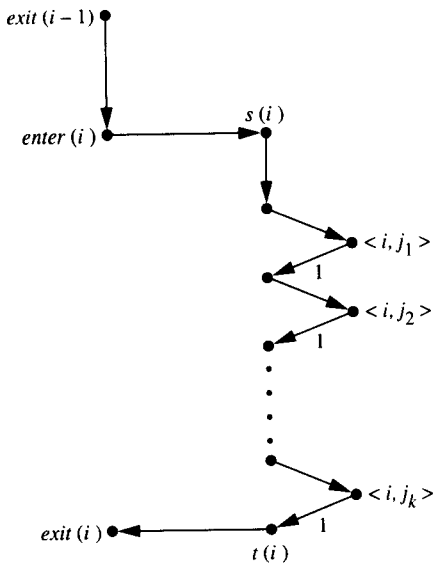


FIGURE 10.8
The component G_i, corresponding to an input g_i, such that g_i appears as input to $g_{j_1}, g_{j_2}, \ldots, g_{j_k}$.

Suppose now that g_i is not an input node. Then, we must have $g_i = \neg(g_j \vee g_k)$, for some $j, k < i$. As before, let $g_{j_1}, g_{j_2}, \ldots, g_{j_k}$ be the nodes that have g_i as an input. The component G_i associated with g_i is shown in Fig. 10.9. A number beside an arc denotes that arc's order in the adjacency list of its vertex. Since $j_1, \ldots, j_k$ are each larger than i, the components $G_1, \ldots, G_k$ will appear later in the chain, such that G_{j_l} contains the vertex $\langle i, j_l \rangle$ for $1 \leq l \leq k$. In addition, the vertices $\langle j, i \rangle$ and $\langle k, i \rangle$ have already appeared in the components G_j and G_k. These facts are illustrated in Fig. 10.10. The overall structure of G is shown in Fig. 10.11. Remember that the components G_i's share vertices, a fact that is not illustrated in the latter figure.

Let s = enter*(1)*. We are ready to prove the following claim.

Claim: An ordered depth-first traversal of G starting at s induces a numbering such that, for each $1 \leq i \leq n$, $dfs(s(i)) < dfs(t(i))$ if and only if the value of node g_i is equal to 1.

Proof of the Claim: We prove a slightly stronger result than the one claimed. Since our proof will use induction, we give a precise statement of the induction hypothesis.

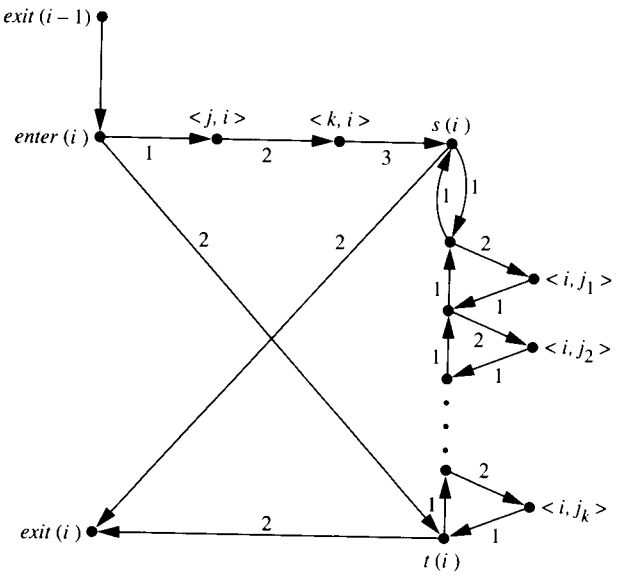


FIGURE 10.9
The component G_i associated with an internal node g_i of the circuit, such that $g_i = \neg(g_j \vee g_k)$ and g_i appears as input to $g_{j_1}, g_{j_2}, \cdots, g_{j_k}$.

Induction Hypothesis: If the value of node g_i is equal to 1, then $dfs(s(i)) < dfs(t(i))$ and, on $enter(i)$ being visited, the path taken from $s(i)$ to $t(i)$ must be as shown in Fig. 10.12. If the value of node g_i is equal to 0, then $dfs(s(i)) > dfs(t(i))$ and, on $enter(i)$ being visited, the path taken from $t(i)$ to $s(i)$ must be as shown in Fig. 10.13.

The base case $i = 1$ corresponds to the input node g_1. Clearly, the depth-first traversal of G_1 starting at s forces $dfs(s(1)) < dfs(t(1))$ and the appropriate path to be taken (since the value of g_1 is 1). Hence, the induction hypothesis holds for the base case.

Assume that the induction hypothesis holds for all indices less than i. Let $g_i = \neg g(g_j \vee g_k)$ (if g_i is an input node, then the proof is simple). If the value of g_i is equal to 1, then we must have $g_j = g_k = 0$. By the induction hypothesis, we have $dfs(s(j)) > dfs(t(j))$, and the path taken on $enter(j)$ being visited follows the pattern indicated in Fig. 10.13. In particular, vertex $\langle j, i \rangle$ is not visited during this traversal. Similarly, vertex $\langle k, i \rangle$ is not visited during the intial traversal of G_k. Therefore, on entering component G_i, we must take the path indicated in Fig. 10.12. Hence, the induction hypothesis holds for this case.

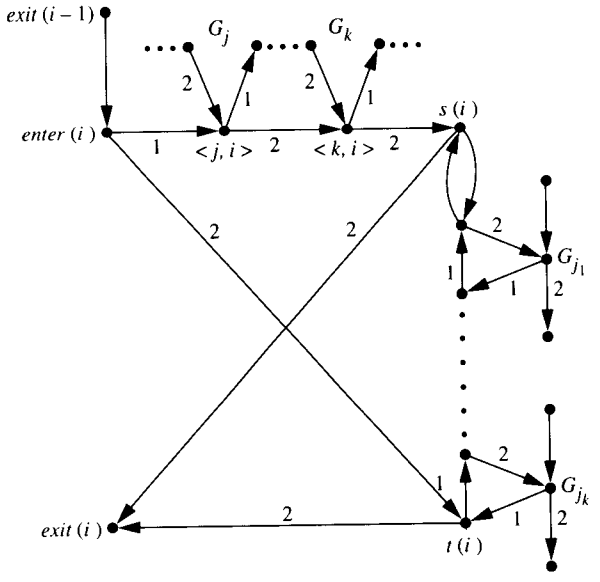


FIGURE 10.10
The component G_i and its relationship to other components. The node g_i is defined by $g_i = \neg(g_j \vee g_k)$, and g_i appears as input in $g_{j_1}, \ldots, g_{j_k}$.

If the value of g_i is equal to 0, then either $g_j = 1$ or $g_k = 1$. Suppose $g_j = 1$. By the induction hypothesis, the path taken during the traversal of G_j must follow the pattern indicated in Fig. 10.12. In particular, vertex $\langle j, i \rangle$ is visited during the initial traversal of G_j. Therefore, on visiting *enter*(i), we must follow the path indicated in Fig. 10.13, and the induction hypothesis holds in this case as well. This result establishes the claim.

Let $u = s(n)$ and $v = s(n)$. Using the claim, $g_n = 1$ if and only if $dfs(u) < dfs(v)$, and the proof of the theorem is complete, since the construction can be carried out easily by an *NC*-algorithm. □

The Maximum Flow Problem. Our next *P*-completeness result concerns the MAXFLOW problem. Before establishing this result, we need to introduce a well-known characterization of maximum flow in terms of augmenting paths.

Let $N = (V, A)$ be a network with s and t as the source and the sink, respectively. The capacity of an edge (u, v) will be denoted by $c(u, v)$. Let f be a flow in N. We would like to determine whether it is possible to augment the

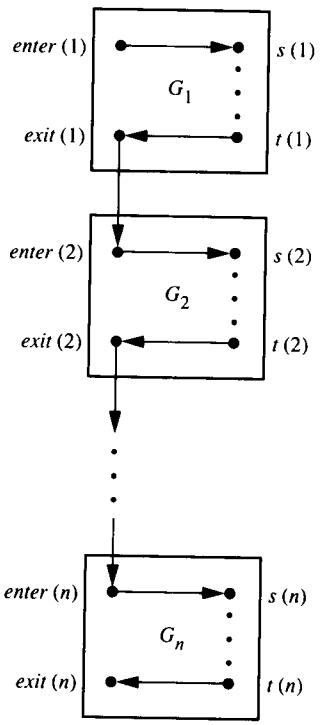


FIGURE 10.11
The structure of the overall graph G associated with a circuit C. Various components may share vertices. An arc inside a box may represent a path between *enter*(i) and $s(i)$.

given flow f such that we preserve the capacity constraint on each arc $e \in A$ and the conservation constraint at each node $v \in V - \{s, t\}$. Such an augmentation is possible if we can find an augmenting path defined as follows.

An **augmenting path** with respect to a given flow f in a network N is an undirected path Q from s to t such that each pair of successive nodes i and j on Q must satisfy one of the following conditions:

1. $(i, j) \in A$ and $f(i, j) < c(i, j)$; such an edge is called a **forward edge.**
2. $(j, i) \in A$ and $f(j, i) > 0$; in this case, the edge (i, j) is called a **backward edge.**

The existence of an augmenting path with respect to the given flow f implies that more flow can be pushed from s to t while the necessary constraints are preserved, as illustrated by the following example.

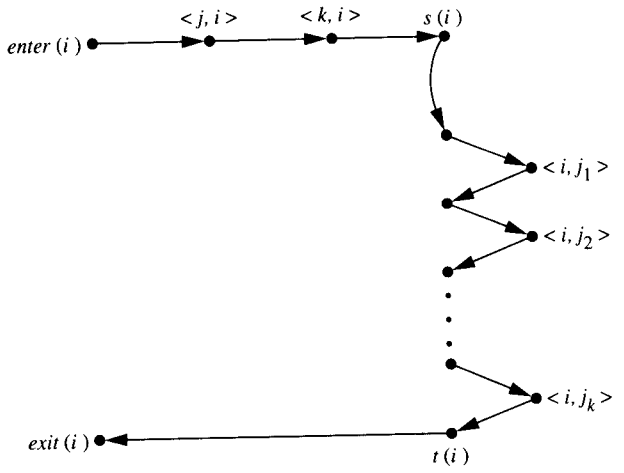


FIGURE 10.12
The path taken on visiting *enter*(*i*), whenever g_i has the value 1.

EXAMPLE 10.9:

Figure 10.14(a) shows a network N, together with the edge capacities, and a given flow f. The notation a/b means that the edge capacity is a and the value of the flow is b. The network N has the augmenting path (s, u, v, w, t) whose edges are all forward, except for (v, u), which is a backward edge. Using this augmenting path, we can increase the value of the flow by 1 as indicated in Fig. 10.14(b). There is no augmenting path with respect to the new flow in N, which implies that the flow is maximum. □

We state without proof the following fundamental theorem.

Theorem 10.15: *A flow f in a network $N = (V, A)$ is maximum if and only if there is no augmenting path with respect to f.* □

Theorem 10.15 can be used as the basis of a polynomial-time sequential algorithm to compute the maximum flow in an arbitrary network. Refer to the bibliographic notes for the details.

We are now ready for our next theorem.

Theorem 10.16: *MAXFLOW is P-complete.*

Proof: We have already observed that MAXFLOW is in P. We now describe an NC-reduction of MCVP2 to MAXFLOW.

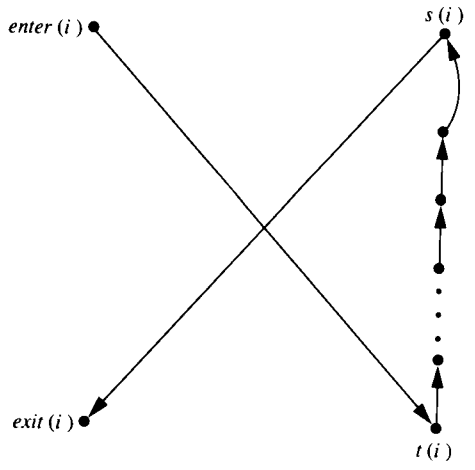


FIGURE 10.13
The path taken on visting *enter(i)*, whenever g_i has the value 0.

Let a circuit C and a specified set of inputs be an instance of MCVP2. We shall construct a network $N = (V, A)$ such that the maximum flow in N is odd if and only if the output of C is equal to 1. In addition, N can be constructed from C by an NC-algorithm.

To make the description of N simpler, we denote the sequence defining C by $C = \langle g_n, \ldots, g_0 \rangle$, where g_0 is the output OR gate. The structure of the network N will be almost identical to that of the directed acyclic graph representing the circuit C, except for two additional vertices s and t and the arcs incident on them.

More precisely, the set V of vertices consists of $\{0, 1, \ldots, n\} \cup \{s, t\}$, where s and t are the two distinguished vertices representing the source and the sink of N, respectively. Vertex i corresponds to node g_i of C, for $0 \leq i \leq n$. The set A of arcs is defined as follows:

1. For each input node g_i of C, we have two arcs (s, i) and (i, s) with the following capacities: $c(i, s) = 2^i$ and

$$c(s, i) = \begin{cases} 2^i \text{ if } g_i = 1; \\ 0 \text{ otherwise.} \end{cases}$$

Figure 10.15(a) illustrates the arcs corresponding to the two inputs g_i and g_j.

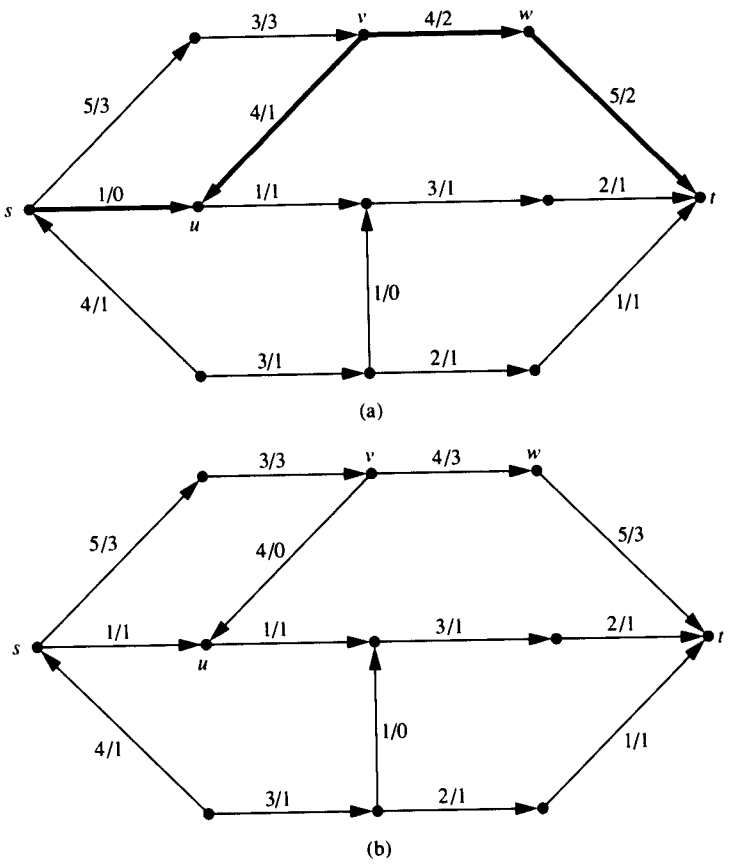


FIGURE 10.14
Network and flows for Example 10.9. (a) A network with the edge capacities and a given flow. The notation *a*/*b* next to an arc indicates that the capacity of the arc is *a*, and the flow through the arc is *b*. (b) The flow after the augmenting path (s, u, v, w, t) is used.

2. For each AND gate—say, $g_i = g_j \wedge g_k$—we have three arcs (j, i), (k, i) and (i, t). Their capacities are $c(j, i) = 2^j$, $c(k, i) = 2^k$, and $c(i, t) = 2^j + 2^k - d2^i$, where d is the fan-out (≤ 2) of gate g_i. Figure 10.15(b) illustrates the arcs corresponding to an AND gate.
3. For each OR gate, say $g_i = g_j \vee g_k$, we have three arcs (j, i), (k, i) and (i, s) whose capacities are $c(j, i) = 2^j$, $c(k, i) = 2^k$ and $c(i, s) = 2^j + 2^k - d2^i$, where d is the fan-out of g_i. Fig. 10.15(c) illustrates this case.

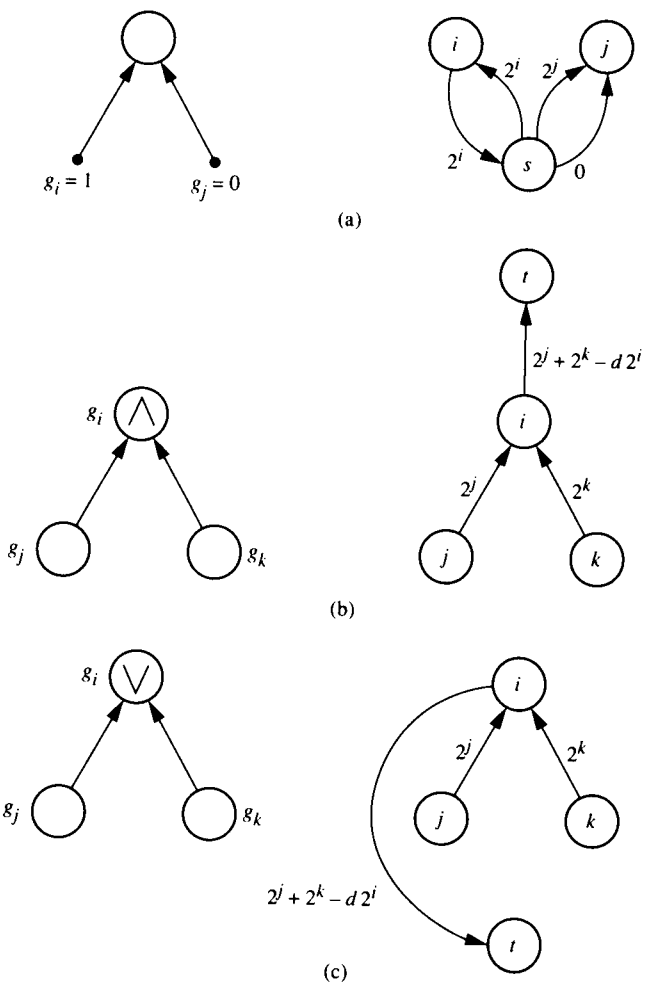


FIGURE 10.15
Arcs corresponding to (a) the input nodes g_i and g_j, (b) an AND gate, and (c) OR gate.

4. Finally, we have the arc $(0, t) \in A$ such that $c(0, t) = 1$.

An example of a circuit and its corresponding network are shown in Figure 10.16.

Constructing the network $N = (V, A)$ consists essentially of replacing each gate g_i of C by at most three arcs in N. Clearly, such a construction can

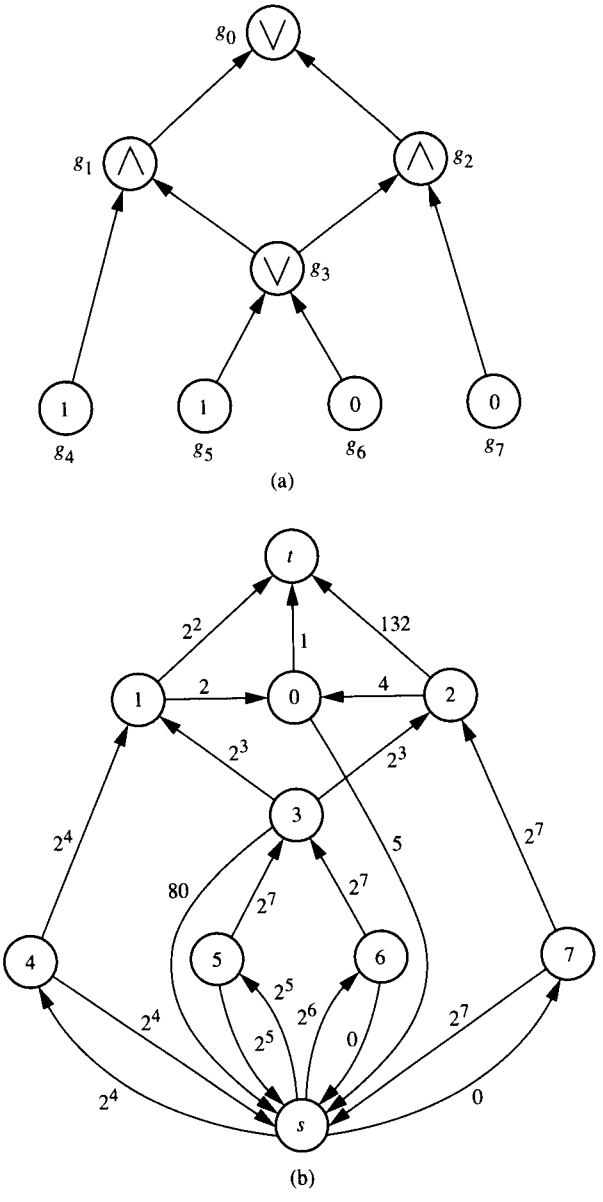


FIGURE 10.16
A circuit and its corresponding network. (a) A Boolean circuit describing an instance of MCVP2. (b) The network corresponding to the circuit shown in (a).

be carried out by an NC-algorithm. The rest of the proof will be completed by establishment of the following claim.

Claim: The maximum flow in $N = (V,A)$ is odd if and only if the output of the circuit C is equal to 1.

Proof of the Claim: We shall introduce a function f on the arcs of N; later we show that f is a maximum flow in N. The function f is defined as follows:

1. For each input node g_i, let $f(s, i) = c(s, i)$. Recall that $c(s, i) = 2^i$ or 0, depending on whether $g_i = 1$ or 0. As for the arc (i, s), set $f(i, s) = f(s, i)$ if g_i does not appear as the input of any gate; otherwise, set $f(i, s) = 0$.
2. For each arc $(i, j) \in A$ such that $i, j \notin \{s, t\}$, set $f(i, j) = 2^i$ if the value of g_i is equal to 1; otherwise, set $f(i, j) = 0$.
3. For each AND gate $g_i = g_j \wedge g_k$, set $f(i, t) = c(i, t)$ if g_i is equal to 1; otherwise, set $f(i, t) = f(j, i) + f(k, i)$.
4. For each OR gate $g_i = g_j \vee g_k$, set $f(i, s) = f(j, i) + f(k, i) - d2^i$ if g_i is equal to 1; otherwise, set $f(i, s) = 0$. As before, d denotes the fan-out of g_i.
5. Finally, set $f(0, t) = 1$ if g_0 is equal to 1 and 0 otherwise.

The function f defines a flow in N. The capacity constraint is obviously satisfied for each arc in N. It is simple to verify that the conservation constraints are satisfied as well (Fig. 10.17 illustrates the two cases of AND and OR gates).

The flow f will always define a maximum flow in N. Suppose that it does not. Then, there exists an augmenting path Q in N relative to f. The path Q must start with a backward edge, since all arcs (s, i) are saturated and must end with a forward arc, since there are no arcs out of t. Hence, we must have three consecutive vertices j, i and k on Q such that (j, i) is a backward edge and (i, k) is a forward edge, as shown in Fig. 10.18.

We finish the proof by showing that such a configuration cannot possibly exit. There are three possible cases:

1. g_i is an input node. Hence, $j = s$, which implies that $f(i, s) = 0$, a contradiction.
2. g_i is an AND gate. Notice that (i, j) and (i, k) are two arcs coming out of vertex i, and $j \neq t$. Hence, g_i is an input to g_j. The fact that $f(i, j) > 0$ implies that $f(i, k) = c(i, k)$, which is impossible.
3. g_i is an OR gate. The flow f is either 0 on all arcs coming out of an OR gate g_i, or the arcs coming out of g_i are all saturated, except possibly for

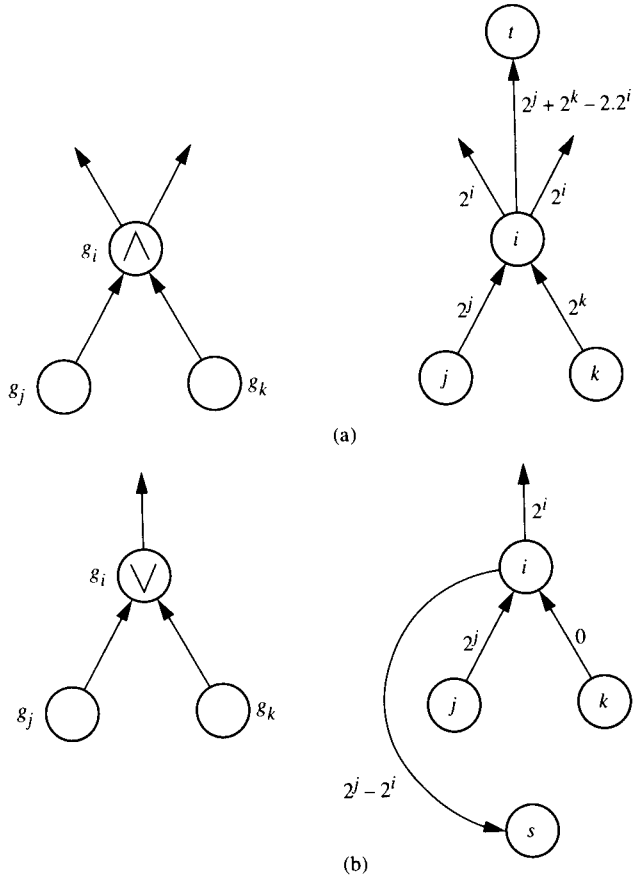


FIGURE 10.17
Flows corresponding to the arcs associated (a) with an AND gate with both inputs equal to 1, and (b) with an OR gate with one input equal to 1.

the arc (i, s) Since $k \neq s$ and $f(i, k) < c(i, k)$, this fact implies that $f(i, j) = 0$, which contradicts the fact that (i, j) is a backward edge.

Therefore, the augmenting path Q cannot exist; hence, f defines a maximum flow in N. The parity of $val(f)$ depends on only $f(0, t)$, since all other arcs into t are assigned flows of even parity. It follows that the ouput of the circuit C is equal to 1 if and only if the maximum flow of C is odd; hence, MAXFLOW is P-complete. □

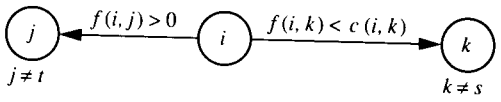


FIGURE 10.18
Three consecutive vertices as they appear in a possible augmenting path.

Linear Inequalities. We show now that the problem of determining the feasibility of a set of linear inequalities (the problem LI introduced earlier) is *P*-complete.

Theorem 10.17: *LI is P-complete.*

Proof: The fact that LI is in *P* follows from our discussions in Section 10.5.4. We now show an *NC*-reduction of CVP to LI.

Let $C = \langle g_1, \ldots, g_n \rangle$, together with a specified set of inputs, be an instance of CVP. With each node g_i of C, we associate a variable x_i and a set of inequalities as follows:

1. If g_i is an input node, then form the equation $x_i = 1$ if the value of g_i is 1, and form $x_i = 0$ otherwise.
2. If g_i is an AND gate $g_i = g_j \wedge g_k$, then form the inequalities

$$\begin{aligned} -x_i &\leq 0 \\ x_i - x_j &\leq 0 \\ x_i - x_k &\leq 0 \\ x_j + x_k - x_i &\leq 1. \end{aligned}$$

3. If g_i is an OR gate $g_i = g_j \vee g_k$, then form the inequalities

$$\begin{aligned} x_i &\leq 1 \\ x_j - x_i &\leq 0 \\ x_k - x_i &\leq 0 \\ x_i - x_j - x_k &\leq 0. \end{aligned}$$

A set of inequalities corresponding to a node g_i forces x_i to be 0 or 1 whenever the inputs are 0 or 1. Add $x_n \leq 1$ and $-x_n \leq -1$ to the preceding inequalities, where g_n is the output gate. The output of the circuit is 1 if and only if $x_n = 1$, and hence if and only if the corresponding linear system of inequalities has a feasible solution. Since CVP is *P*-complete, it follows that LI is *P*-complete. □

10.6 Summary

We established, in this chapter, several inherent limitations on PRAM algorithms. As a result, several specific functions were shown to require logarithmic or near logarithmic number of steps under fairly general assumptions. In particular, we saw that the simple problem of computing the Boolean OR of n bits requires $\Omega(\log n)$ time on the ideal CREW PRAM, regardless of the number of processors available, whereas it can be solved in $O(1)$ time, using n processors, on the common CRCW PRAM. We also saw that, on the EREW PRAM with a relatively large number of processors, the problem of searching a sorted list cannot be handled significantly faster than it can be handled on a single processor. In addition, using an algebraic technique for deriving lower bounds on bounded-depth circuits, we showed that the parity function on n bits requires $\Omega(\log n/\log\log n)$ time on the priority CRCW PRAM, using any polynomial number of processors. Finally, we addressed the issue of whether there exist extremely fast parallel algorithms at a reasonable cost for solving several important problems, such as maximum flow, linear programming, and depth-first search. The *NC*-theory was used to provide a strong evidence on the negative side.

The topics covered in this chapter touch on a few elements of **parallel compexity theory.** Our presentation has been substantially tilted toward the PRAM model, in the same spirit as in the rest of the book. To put things in a better perspective, we note that three fundamental notions are closely related: **parallel time, sequential space,** and **circuit depth.** The relationships among these measures form the basis of a beautiful theory of parallel complexity. The bibliographic notes contain pointers to several references that discuss this theory in depth. Here, we will sketch a result from this theory.

We define the class NC^k to be the set of functions computable by a uniform family of circuits $\{C_n\}$ such that the size of C_n is polynomial in n and its depth is $O(\log^k n)$, for $k \geq 1$. Similarly, we define $EREW^k$, $CREW^k$, and $CRCW^k$ to be the classes of functions computable by the EREW, CREW, CRCW PRAMs, respectively, in $O(\log^k n)$ time, using a polynomial number of processors. Then, we have the following relationship among these classes:

$$NC^k \subseteq EREW^k \subseteq CREW^k \subseteq CRCW^k \subseteq NC^{k+1}.$$

Hence, the class *NC* could have been defined by $NC = \cup_{k\geq 1} NC^k$, which does not involve the PRAM model at all. In fact, the class *NC* remains invariant under a variety of parallel models; hence, its complexity is of fundamental importance. Determining the exact relationships among these classes remains a major open research goal.

Exercises

10.1. Describe in detail how a concurrent-write instruction of a p-processor priority CRCW PRAM can be implemented on a p-processor EREW PRAM in $O(\log p)$ time.

10.2. Our simulation of the priority CRCW PRAM on the EREW PRAM has been expressed in the time-processors framework. Consider using the time-work framework. Given a priority CRCW algorithm running in $O(T(n))$ time and using a total of $W(n)$ operations, what are the corresponding time and total number of operations required by the simulating EREW PRAM?

10.3. Given a set of positive integers $x_1, x_2, \ldots, x_n$ stored in the first n cells of the global memory of an arbitrary CRCW PRAM, the **element-distinctness problem** is to determine whether there exist $i \neq j$ such that $x_i = x_j$. Show how to solve this problem in $O(1)$ time, using n processors. *Hint*: Exploit the fact that the x_i's are positive integers.

10.4. You are given a common CRCW PRAM with $p \log p$ processors such that each of the global locations $M_1, M_2, \ldots, M_p$ contains a pair $\langle j_i, \alpha_i \rangle$, for $1 \leq i \leq p$. Write down the algorithm to be executed by each processor P_i, such that α_i will be written into location j_i in $O(1)$ time. If several j_i's are equal, the smallest indexed α_i is written into location j_i.

10.5. Each processor P_i of a p-processor arbitrary CRCW PRAM has a write instruction into a certain global memory location M_{j_i}, for $1 \leq i \leq p$. Write down the algorithm to be executed by each P_i to simulate a concurrent write on the priority CRCW PRAM. Assume that the memory locations $M_0, M_1, \ldots, M_{p-1}$ are special and contain 0 initially.

Exercises 10.6 through 10.8 shed light on the relative powers of the CRCW models as a function of the number m of shared memory locations when the number of processors is held fixed at n. A particular model is denoted by its name, followed by the number of shared memory locations in parentheses; priority(1) and common(m) are examples.

10.6. The **1-color minimization** problem is described as follows. Given a set of n processors P_i each having a color $x_i \in \{0, 1\}$ in its local memory, each processor P_i is supposed to compute the value $a_i \in \{0, 1\}$ such that $a_i = 1$ if and only if P_i is the lowest-indexed processor with $x_i = 1$. We have already seen how to solve this problem in $O(1)$ time on the common CRCW, when no restrictions are placed on the size of the shared memory. Generalize this algorithm (Exercise 2.13) to show that the 1-color minimization problem can be solved in $O\left(\frac{\log n}{\log (m+1)}\right)$ time on common(m).

10.7. Use Exercise 10.6 to show that common(m) can simulate one step of priority(m) in $O(\log n)$ steps.

10.8. *Use an adversary argument to show that the 1-color minimization problem introduced in Exercise 10.6 requires at least $\frac{\log(n+1)-1}{\log(m+1)}$ steps on arbitrary(m).

10.9. Show how to reduce the problem of simulating the priority CRCW PRAM on the arbitrary CRCW PRAM to integer sorting in $O(\log n)$ time using $O(n)$ operations.

10.10. Give an example of a Boolean function of n variables that does not have a critical input and whose computation requires $\Omega(\log n)$ time on the CREW PRAM with any number of processors.

10.11. Prove Lemma 10.3.

10.12. Where did we need the exclusive-write assumption in the proof of Lemma 10.3?

10.13. Establish the correctness of Corollary 10.3.

10.14. The purpose of this exercise is to show that the Boolean OR function of n variables can be computed by an n-processor CREW PRAM in $\leq \log_{2.618} n + O(1)$ steps, which is less than $\log_2 n$.

Let the input bits $x_1, \ldots, x_n$ be stored in $M(1), \ldots, M(n)$ of the global memory, and let $n = F_{2T+1}$, where F_i is the ith Fibonacci number. Recall that the Fibonacci numbers are defined by $F_0 = 0, F_1 = 1$ and $F_{m+2} = F_{m+1} + F_m$, for $m \geq 0$. Each processor P_i uses two variables y_i and t, which are initially set to 0. The algorithm executed by P_i is the following:

if $i + F_{2t} \leq n$ **then** set $y_i := y_i + M(i + F_{2t})$
if ($i > F_{2t+1}$ and $y_i = 1$) **then** set $M(i - F_{2t+1}) := 1$

a. Show that, just before step t, we have $y_i = x_i + x_{i+1} + \cdots + x_{i+F_{2t}-1}$ and $M(i) = x_i + x_{i+1} + \cdots + x_{i+F_{2t+1}-1}$, for $1 \leq i \leq n$.

b. Deduce that the algorithm uses at most $\log_{2.618} n + O(1)$ steps.

10.15. Consider the computation of x^n, where x is an input data stored in a location of the global memory.

a. Show that x^n can be computed in one step on the ideal PRAM model.

b. Show that $\Omega(\log n)$ steps are required by the standard arithmetic CREW PRAM. Assume that each step consists of a read instruction, or a write instruction, or an arithmetic operation from $\{+, -, \times, \div\}$. *Hint*: Show that any function computed at the kth step is of degree $\leq 2^k$.

10.16. Given a string w of length n and a set $S \subset \{1, \ldots n\}$, the notation $w^{(S)}$ indicates the string w with all the bits in S flipped. A Boolean function f on n variables is **sensitive** to x_i on w if $f(w) \neq f(w^{(i)})$. The **sensitivity**

$s_w(f)$ of f on w is the number of indices i such that f is sensitive to x_i on w. Define the **sensitivity** $s(f)$ of f to be $s(f) = \max_w \{s_w(f)\}$. Note that, if f has a critical input then $s(f) = n$.

a. Give an example of a function f that does not have a critical input but that satisfies $s(f) = \Omega(n)$.

b. Show that the time required to compute an arbitrary Boolean function f on the ideal CREW PRAM must satisfy $T = \Omega(\log s(f))$.

10.17. Let f be a Boolean function of n variables, and let $S \subset \{1, \ldots, n\}$. Given a string w of n bits, f is **sensitive** to S on w if $f(w) \neq f(w^{(S)})$, where $w^{(S)}$ is the same as w with the bits in S flipped. The **block sensitivity** of w, denoted by $bs_w(f)$, is the largest t such that there exist disjoint sets $S_1, S_2, \ldots, S_t$ such that, for all $1 \leq i \leq t$, f is sensitive to S_i on w. The *block sensitivity* of f is given by $bs(f) = \max_w bs_w(f)$. Show that it takes $\Omega(\log bs(f))$ time to compute f on the ideal CREW PRAM. *Hint*: Define a new function $f'(y_1, \ldots, y_t)$ where $t = bs(f)$.

10.18. Lemma 10.6 applies to the Boolean OR function with I_i replaced by the critical input I, and I_{i-1} replaced by $I(i)$. Can you use this lemma to conclude that $|L(M, t)| \leq |L(M, t-1)| + 2|K(P, t)|$, and hence derive a simpler proof for Theorem 10.5? Explain your answer.

10.19. **a.** Show that almost all Boolean functions require unbounded fan-in circuits of size $\Omega(2^{n/2})$. *Hint*: Determine an upper bound on the number of circuits with s gates.

b. Deduce that restricted CRCW PRAMs with a polynomial number of processors require exponential time to compute almost all Boolean functions.

c. What are the resources required by the ideal CRCW PRAMs?

10.20. Given a set of n bits $x_1, \ldots, x_n$, show that the problem of computing the binary representation of the sum $x_1 + \cdots + x_n$ requires $\Omega\left(\frac{\log n}{\log \log n}\right)$ time on the restricted CRCW PRAM with a polynomial number of processors.

10.21. Given two n-bit binary numbers, show that the problem of computing the binary representation of their product requires $\Omega\left(\frac{\log n}{\log \log n}\right)$ steps on the restricted priority CRCW PRAM, using a polynomial number of processors.

10.22. Consider an instance of the monotone circuit-value problem $C = \langle g_1, g_2, \ldots, g_n \rangle$, where each g_i is either 0 or 1, or $g_i = g_{i_1} \wedge g_{i_2} \wedge \cdots \wedge g_{i_{j(i)}}$ or $g_i = g_{i_1} \vee g_{i_2} \vee \cdots \vee g_{i_{j(i)}}$, for $i_1, \ldots, i_{j(i)} < i$. Let m be the total number of edges in the corresponding directed acyclic graph. Develop a common CRCW algorithm to compute g_n in $O\left(\frac{m}{p} + n\right)$ time, using p processors.

10.23. Develop a PRAM algorithm to compute the DFS numbers of the vertices of a directed graph $G = (V, E)$ where the graph is given by its adjacency lists as in ordered DFS. Your algorithm must run in $O\left(\frac{|E|}{p} + |V|\right)$ time, using p processors.

10.24. Given an *NC*-algorithm to solve the ordered DFS decision problem, derive an *NC*-algorithm to solve the ordered DFS problem.

10.25. Develop an *NC*-reduction from CVP to MCVP.

10.26. Prove Lemma 10.12.

10.27. Prove that the (NOR-CVP) is *P*-complete.

10.28. Prove that the fan-out 2 (MCVP2) is *P*-complete.

10.29. Given an arithmetic circuit defined by a sequence $AC = \langle a_1, \ldots, a_n \rangle$ such that each $a_i = 0$ or 1, or $a_i = a_j \pm a_k$ or $a_i = a_j \times a_k$ for some $j, k < i$, we are interested in determining whether or not the value of g_n is equal to 1. Show that this problem is *P*-complete.

10.30. The **lexicographically first maximal-clique (LFMC)** problem is stated as follows: Given an undirected graph $G = (V, E)$ with an ordering on the vertices and a vertex $v \in V$, is v in the LFMC clique of G? The graph G shown in Figure 10.19 has $\{v_1, v_2, v_3\}$ as the LFMC. Show that LFMC is *P*-complete. *Hint*: Use MCVP.

10.31. Given a finite set X with a binary operation $\bigcirc$, an initial subset $I \subseteq X$, and an element $x \in X$, the **generability problem** is to determine whether x is in the closure (under the operation $\bigcirc$) of I. Show that the generability problem is *P*-complete.

10.32. Given a path system $P = (X, R, S, T)$, where $S \subseteq X$, $T \subseteq X$, and $R \subseteq X \times X \times X$, the **PATH problem** is to determine whether there exists an admissible node in S. A node x is *admissible* if and only if $x \in T$ or there exist $y, z \in X$ such that $(x, y, z) \in R$ and both y and z are admissible. Show that the **PATH problem** is *P*-complete. *Hint*: Use a reduction from generability.

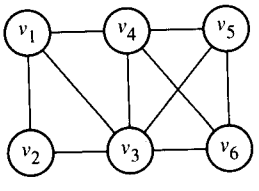


FIGURE 10.19
A graph whose first lexicographically maximal clique consists of the vertices v_1, v_2, and v_3.

10.33. * The **planar circuit value problem (PCVP)** is described as follows: Given a circuit $C = \langle g_1, \ldots, g_n \rangle$ representing a *planar* circuit, determine whether the output is equal to 1. Show that PCVP is P-complete.

Bibliographic Notes

Early work on the relations among the different PRAM models was reported in [13, 26, 39]. Our formulation of the leftmost-prisoner problem, as well as the simulation of a p-processor priority CRCW PRAM by a p-processor common CRCW PRAM in $O\left(\frac{\log p}{\log \log p}\right)$ time, is taken from Fich, Radge, and Widgerson [15]. This separation between the common and the priority CRCW PRAMs is optimal, since the element-distinctness problem, although solvable in $O(1)$ time on the priority CRCW PRAM, requires $\Omega\left(\frac{\log p}{\log \log p}\right)$ time on the common CRCW, as shown by Boppana in [5]. That a $p \log p$-processor common CRCW can simulate a p-processor priority CRCW in $O(1)$ time, and that the arbitrary CRCW can simulate the priority CRCW in $O(\log \log p)$ time, have been shown by Chlebus, Diks, Hagerup, and Radzik in [7]. More efficient randomized simulations appear in [17]. The techniques described to prove lower bounds for the CREW PRAM were introduced by Cook, Dwork, and Reischuk in [10]. Exercise 10.14 also is taken from there. The generalization mentioned in Exercise 10.17, as well as a matching upper bound, were established in [30]. Sharper results appear in [11], [27], and [33]. Our lower-bound proof for searching a sorted list on the EREW PRAM follows Snir [37]. The simulation results established in the text for CRCW PRAMs and unbounded fan-in circuits appeared in [38] by Stockmeyer and Vishkin. Considerable work has been done on lower bounds for bounded-depth circuits since the initial work that appeared in [16]. The interested reader should consult the paper by Boppana and Sipser [6] for more results and references. The algebraic approach described in the text was introduced by Smolensky [36]; our presentation follows [6]. The lower bound on computing the parity function shown for the restricted priority CRCW PRAM holds even on the ideal priority CRCW PRAM. For the proof, see the paper by Beame and Hastad [4].

The original motivation behind the P-completeness results was to study the relative power of the two classes P and POLYLOGSPACE; hence, these results play a role similar to that of the NP-completeness results relating P and NP. Here, POLYLOGSPACE is the class of all languages recognizable in $O(\log^k n)$ space, for some fixed constant k, and the notion of reducibility used is the log-space reducibility. It is easy to verify that the reductions we presented can be achieved in $O(\log n)$ space. Hence, our P-complete problems are all log-space complete for P. The class NC is named after Nick Pippenger (Nick's Class), who introduced the class of languages recognizable in polynomial-sized and polylog-depth circuits [8, 34]. There is a close relation between space and parallel time that leads to the **parallel-computation thesis,** which essentially states that space and parallel time are polynomially related. Such a relationship has been shown for several parallel models, including the PRAM model. For further reading on this topic, see [8, 19, 32].

The circuit-value problem was shown to be log-space complete for P by Ladner [28]; Jones and Laaser [23] added a significant number of problems to the list of complete problems for P, including generability (Exercise 10.31) and path systems (Exercise 10.32). Our proof of the P-completeness of ordered depth-first search was developed

by Reif [35]; our MAXFLOW proof was given by Goldschlager, Shaw, and Staples [20]. Polynomial-time algorithms for the maximum flow problem are described in [31]. Our *P*-completeness proof of determining whether a set of LI is feasible is attributed in [22] to S. Cook. Polynomial-time algorithms for linear programming appeared in [24, 25]. The works [1, 2, 3, 12, 21, 29] contain additional *P*-complete problems.

Solutions to Exercises 10.6 through 10.8 and several other simulation results on CRCW PRAMs with limited amounts of memory can be found in [14]. Exercise 10.9 was suggested by Y. Matias. The solution to Exercise 10.19 is presented in [4], and the solutions to Exercises 10.22 and 10.23 are given in [40]. Exercise 10.30 is taken from [9] and Exercise 10.33 is taken from [18].

References

1. Anderson, R., and E. Mayr. Parallelism and greedy algorithms, Volume 4 of *Advances in Computing Research*, F. P. Preparata ed., JAI Press Inc., Greenwich, CT, 1987.
2. Anderson, R., and E. W. Mayr. Parallelism and the maximal path problem. *Information Processing Letters*, 24(2):121–126, 1978.
3. Avenhaus, J., and K. Madlener. The Nielson reduction and P-complete problems in free groups. *Theoretical Computer Science*, 32(1,2):61–76, 1984.
4. Beame, P., and J. Hastad. Optimal bounds for decision problems on the CRCW PRAM. *JACM*, 36(3):643–670, 1989.
5. Boppana, R. B. Optimal separations between concurrent-write parallel machines. In *Proceedings Twentieth Annual ACM Symposium on Theory of Computing*, Seattle, WA, 1989, pp. 320–326.
6. Boppana, R. B., and M. Sipser. The complexity of finite functions. *Handbook of Theoretical Computer Science*, Volume A: Algorithms and Complexity, J. Van Leuwen ed., MIT Press, Cambridge, MA, 1990.
7. Chlebus, B. S., K. Diks, T. Hagerup, and T. Radzik. Efficient simulations between concurrent-read concurrent-write PRAM models. In *Proceedings Thirteenth Symposium Math. Found. Comp. Sci.*, Carlsbad, Czechoslovakia, 1988, pp. 231–239.
8. Cook, S. A. Towards a complexity theory of synchronous parallel computation. In *L'Enseignement Mathematique*, 27(1,2):99–124, 1981.
9. Cook, S. A. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64(1–3):2–22, 1985.
10. Cook, S. A, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM J. Computing*, 15(1):87–97, 1986.
11. Dietzfelbinger, M., M. Kutylowski, and R. Reischuk. Exact lower time bounds for computing Boolean functions on CREW PRAMs. In *Proceedings of the Second Annual ACM Symposium on Parallel Algorithms and Architectures*, Island of Crete, Greece, 1990, pp. 125–135.
12. Dwork, C., P. Kanellakis, and J. Mitchen. On the sequential nature of unification. *Journal of Logic Programming*, 1(1):35–50, 1984.
13. Eckstein, D. M. Simultaneous memory access. Technical Report TR 79-6, Computer Science Department, Iowa State University, Ames, IA, 1979.

14. Fich, F. E., P. Ragde, and A. Widgerson. Relations between concurrent-write models of parallel computations. *SIAM J. Computing*, 17(3):606-627, 1988.

15. Fich, F. E., P. Ragde, and A. Wigderson. Simulations among concurrent-write PRAMs. *Algorithmica*, 3(1):43-51, 1988.

16. Furst, M., J. B. Saxe, and M. Sipser. Parity, circuits and the polynomial time hierarchy. *Math. Systems Theory*, 17(1):13-28, 1984.

17. Gil, J., and Y. Matias. Leaders election without a conflict resolution rule-fast and efficient randomized simulations among CRCW PRAM. Private communication, 1991.

18. Goldschlager, L. M. The monotone and planar circuit value problems are log-space complete for P. *SIGACT News*, 9(2):25-29, 1977.

19. Goldschlager, L. M. A universal interconnection pattern for parallel computers. *JACM*, 29(4):1073-1086, 1982.

20. Goldschlager, L. M., R. A. Shaw, and J. Staples. The maximum flow problem is log space complete for P. *Theoretical Computer Science*, 21(1):105-111, 1982.

21. Helmbold, D., and E. Mayr. *Fast scheduling algorithms on parallel computers*, Volume 4 of *Advances in Computing Research*, F. P. Preparata ed., JAI Press Inc., Greenwich, CT, 1987.

22. Hoover, H. J., and W. L. Ruzzo. A compendium of problems complete for P. Unpublished manuscript. Updated version with R. Greenlaw appears as TR 91-14, Department of Computer Science, University of New Hampshire, 1991.

23. Jones, N. D., and W. T. Laaser. Complete problems for deterministic polynomial time. *Theoretical Computer Science*, 3:105-117, 1976.

24. Karmarkar, N. A new polynomial time algorithm for linear programming. *Combinatorica*, 4(4):373-395, 1984.

25. Khachian, L. G. A polynomial time algorithm for linear programming. *Soviet Math. Dokl.*, 20(1):191-194, 1979.

26. Kucera, L. Parallel computation and conflicts in memory access. *Information Processing Letters*, 14(2):93-96, 1982.

27. Kutylowski, M. Time complexity of Boolean functions on CREW PRAMs. *SIAM J. Computing*, 20(5):824-833, 1991.

28. Ladner, R. E. The circuit value problem is log space complete for P. *SIGACT News*, 7(1):18-20, 1975.

29. Miyano, S. The lexicographically first maximal subgraph problems: *P*-completeness and *NC* algorithms. In *Proceedings of the 1987 ICALP Conference*, Karlshue, Germany, 1987, pp. 425-434.

30. Nisan, N. CREW PRAMs and decision trees. *SIAM J. Computing*, 20(6):999-1007, 1991.

31. Papadimitriou, C. H., and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Englewood Cliffs, NJ, 1982.

32. Parberry, I. *Parallel Complexity Theory.* Research Notes in Theoretical Computer Science. John Wiley & Sons, New York, 1987.

33. Parberry, I., and P. Yuan Yan. Improved upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM J. Computing*, 20(1):88-99, 1991.

34. Pippenger, N. On simultaneous resource bounds. In *Proceedings Twentieth Annual IEEE Symposium on Foundations of Computer Science*, San Juan, Puerto Rico, 1979, pp. 307–311.

35. Reif, J. H. Depth first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.

36. Smolensky, R. Algebraic methods in the theory of lower bounds for Boolean circuit complexity. In *Proceedings Nineteenth Annual ACM Symposium on Theory of Computing*, New York, NY, 1987, pp. 77–82.

37. Snir, M. On parallel searching. *SIAM J. Computing*, 14(3):688–707, 1985.

38. Stockmeyer, L., and U. Vishkin. Simulation of parallel random access machines by circuits. *SIAM J. Computing*, 13(2):409–422, 1984.

39. Vishkin, U. Implementation of simultaneous memory address access in models that forbid it. *Journal of Algorithms*, 4(1):45–50, 1983.

40. Vitter, J. S., and R. A. Simons. New classes for parallel complexity: A study of unification and other complete problems for *P*. *IEEE Transactions on Computers*, C-35(5):403–418, 1986.

INDEX

1-color minimization problem, 554
2-3 trees, 65–70
2CNF, 258

accelerated cascading, 71–75
adjoint matrix, 427, 479
adversary argument, 187
all nearest neighbors, 302
all nearest smaller values
    (ANSV), 83, 195–196
all pairs shortest paths, 246, 250–252
altitude, 163
AND Boolean function, 37
  see also OR Boolean function
antipodal, 301
approximate median, 489
approximator, 524–528
augmenting path, 474, 544

back substitution algorithm, 36–37
balanced trees, 43–51
basis cycle, 228
Bayes' theorem, 485
Benes networks, 200
Bernoulli trial, 438
biconnected components, 227–239, 255
binary search tree, 466, 488
binary search, 62
binomial distribution, 438–439
binomial theorem, 439
bipartite graph, 255
bitonic sequence, 174–178
bitonic sort, 178–181
block sensitivity, 556
blocks of a graph, 227–239
Boole's inequality, 436
Boolean circuit, 513
  unbounded fan-in, 513–528
Boyer-Moore algorithm, 318, 363
breadth-first search, 257
bridge, 255
broadcasting,
  on the hypercube, 22–23
  on the PRAM, 37

c-cover of a sequence, 156
cascading divide-and-conquer, 161–173
Cayley-Hamilton theorem, 403–404, 427
centroid of a tree, 141
characteristic polynomial, 403, 426–427
Chebyshev bound, 438
Chernoff bounds, 439
Chinese postman problem, 257
Chistov's algorithm, 427–428
circuit value problem, 536–539
circulant matrix, 393, 424–425
Clos network, 199–200
coarsest partitioning problem, 259
coin tossing, 434–435, 436, 437
coloring,
  a cycle, 75–80
  a pseudoforest, 87
communication complexity, 33–35
compacting, 82
comparator network, 173
comparison tree model, 185
connected components, 204–221
convex hull, 56–60, 264–272
convolution, 388–389, 424
cost of a parallel algorithm, 26, 31–32
CRCW PRAM, 15
CRCW PRAM, arbitrary, 15
CRCW PRAM, common, 15
CRCW PRAM, priority, 15
CREW PRAM, 15
critical input for a Boolean function, 503
cut vertex, 255
cycle basis, 227

data parallel algorithms, 26
Davenport-Schinzel sequence, 304
depth-first search 115, 204–205,
    531–532, 540–543
descriptor of a substring, 344–346
determinant, 405, 406
  of a Tutte matrix, 476–478
  of an integer matrix, 479
deterministic sample, 359–360
diameter, of a convex polygon, 301
  of a hypercube, 21
  of a linear array, 17
  of a network, 17
  of a two-dimensional mesh, 19
digital search tree, 340–342
directed cycle, 76, 442–443
discrete Fourier transform, 379–386
discrete Fourier transform, inverse, 385
distribution function, 437
divide-and-conquer, 56–61
dominance counting, 291–296

ear decomposition, 240–245
  open, 240
edge coloring, 256
efficiency, 4



element distinctness problem, 554
elementary event, 435
elementary symmetric functions, 424
embedding of graphs, 38
EREW PRAM, 15
Euclidean norm, 408
Euler array, 129
Euler circuits, 108, 256
Euler partition, 256
Euler tour, 108–118
exclusive OR of cycles, 227
expected value, 437
expression evaluation, 119, 122–125
extended Euclid's algorithm, 415–416
extremal graph theory, 188
extreme point, 265

failure function, 325, 337–339
fan-out-2 monotone circuit value problem, 539
Fast Fourier Transform (FFT) algorithm, 381–385
feasible region corresponding to linear inequalities, 277
Fibonacci numbers, 357
Fibonacci strings, 357
fingerprint functions, 451–456
finitely generated sequence, 487
Floyd-Warshall algorithm, 258–259
fractional independent set, 441–445
fully normal hypercube algorithms, 23

generability problem, 557
generating polynomial of a sequence, 487
greatest common divisor,
  integers, 313
  polynomials, 415–419

hash function, perfect, 361
hashing, 451
high function, 236
Horner's algorithm, 37, 368–369, 372
hypercube, 20–24, 38–39

ideal PRAM, 502–503
incidence matrix,
  directed graphs, 246
  undirected graphs, 207
independent set,
  in a graph, 188
  linked list, 93–96
  of cycles, 227
  see also fractional independent set
inorder traversal, 128, 139, 196
intersection graph, 142
intersection of half-planes, 272–277
interval graph, 142
isolating lemma of a set system, 480
iterative methods for matrix inversion, 409

kernel of a polygon, 306
Knuth-Morris-Pratt algorithm, 318, 324–327
Kruskal's algorithm, 223
Krylov matrix, 374

Lagrange interpolation formula, 400
Las Vegas algorithms, 440
LDU factorization, 369–370
leftmost prisoner problem, 498–499
level of a vertex in trees, 116
lexicographically first maximal clique, 557
line segment, 264
linear order, 61
linear processor array, 17–19
linear programming, 277–279, 540, 552
linear recurrences, 368–375
linear systems, 406
  banded triangular, 370, 376–379
  triangular, 375–376
linear inequalities, 540, 552
list contraction, 102–108
list ranking, 92–108
locus of a node in suffix trees, 343, 353
low function, 236
lower bounds,
  CRCW PRAM, 512–529
  CREW PRAM, 502–508
  EREW PRAM, 508–511
  majority function, 529
  maximum, 188–189
  merging, 189–192
  parity function, 528
  searching, 187–188
  sorting, 192
  unbounded fan-in circuits, 524–528
lower envelope, 286–291, 304
lowest common ancestor, 128–136, 142
LU factorization, 406–407

Markov inequality, 485
matching, in a graph, 474–484
  in a tree, 140
  perfect, 460
matrix inversion, 406, 409
matrix multiplication, 9, 15–16, 19–20, 23–24, 33–35, 247–249
  Boolean, 248–249
matrix powers, 249
matrix vector multiplication, 11–12, 17–19
matrix, lower triangular, 84–85, 375–376
matrix, tridiagonal, 422
maximum flow, 532–534, 540, 543–551
maximum, 71–74
mean value, 437
median, 181
median, approximate, 489
mergesort, 158–160
merging, 61–65, 148–157
message passing, 17
MIMD, 11
minimum spanning trees, 222–227

monotone circuit value problem, 539
Monte Carlo algorithms, 440
multilocation of a point, 285
multiple instruction multiple data (MIMD), 11

NC class, 534–535
network flow, 532–534, 543–545
network model, 16–24
networks, Benes, 200
  Clos, 199–200
  comparator, 173
Newton's identities, 426
Newton's iteration, 411–412, 425
NOR circuit value problem, 539
normal hypercube algorithms, 23
number of descendants, 118

oblivious algorithm, 174
odd-even merge, 181, 198
optimal parallel algorithm, 32
OR Boolean function, 507
  see also AND Boolean function
ordered depth-first search, 531–532, 540–543

P-completeness, 529–552
parallel complexity of a dag, 9
parallel computation thesis, 558
parallel prefix, 53–55, 99
parallel random access machine (PRAM), 11–16
parallel search of a sorted list, 146–148
pardo statement, 27
parentheses matching, 141, 196
parity function, 526–528
partitioning, 61–65
PATH problem, 557
pattern analysis, 319, 327–339
pattern matching, 450–459
perfect hash function, 361
period of a string, 312–315
periodicity lemma, 313
pipelined merge sort, 161–172
pipelining, 65–70, 161–172
planar circuit value problem, 558
planar graph, 305
planar subdivision, 305, 447
  point location, 445–450
  triangulated, 446
plane sweep tree, 280–283
plane sweeping, 279–285
pointer jumping, 52–56
polygon, 265
polygonal chain, 265
polynomial division, 394–397
polynomial evaluation, 368–369, 372, 386, 397–400
polynomial greatest common divisor, 415–419
polynomial identities, 460–464
polynomial interpolation, 386, 400–402
polynomial multiplication, 386–388
postorder traversal, 115–116
PRAM, 11–16
  arbitrary CRCW, 15
  arithmetic, 368
  common CRCW, 15
  comparison, 186–187
  CRCW, 15
  CREW, 15
  EREW, 15
  ideal, 502–503
  priority CRCW, 15
  randomized, 440
  restricted, 516
precedence graphs, 7–9
predecessor of an element in a sorted list, 151, 286
prefix minima, 83, 85, 133
prefix sums, 38, 44–51
  segmented, 83
preorder traversal, 139, 143, 197, 232
Prim's algorithm, 223
primitive root of unity, 385
probability distribution, 437
probability measure, 435
processor allocation, 28–31, 49–51
pseudoforest, 87, 205

QR factorization, 406, 407
quicksort, 196, 465–473

rake operation on a tree, 119
random access machine (RAM), 6
random sampling, 465
random variable, 437
randomized PRAM, 440
range minima problem, 131–136
receive statement, 16
reducibility, 535–536
reflexive transitive closure, 230
regular expressions, 360
remainder sequence, 416
repeated squaring algorithm, 36, 250
residual of an approximation to matrix inverse, 409
restricted PRAM, 516
resultant matrix, 410
ring network, 17–19
ring, 248
RNC class, 534–535
rooting a tree, 114–115
routing, 17
ruler of a linked list, 101
running time of a sequential algorithm, 6

sample of a sorted sequence, 161
sample sort, 470–473
sample space, 435
schedule of a dag, 8–9

selection, 181-184, 197-198
semicircle property, 302
send statement, 16
sensitivity of a Boolean function, 555-556
shared memory model, 9-16
shortest paths, 246, 250-252, 258-259
SIMD, 13
simulation,
  between circuits and CRCWs, 515-523
  priority CRCW on arbitrary CRCW, 501-502
  priority CRCW on common CRCW, 498-501
  priority CRCW on EREW, 496-498
single instruction multiple data (SIMD), 13
Sollin's algorithm, 223-227
sorting, 157-173, 196-199
  bitonic, 174-181
  integers, 79
  mergesort, 158-160
  networks, 173-181
  pipelined mergesort, 160-173
  randomized, 464-473
sparse graphs, 213
speedup, 3
standard deviation, 437
Stirling's formula, 439
string editing, 360-361
string matching with k differences, 362-363
string matching, 318-339, 456-457
  multipattern, 363
  on-line, 352-355
string,
  periodic, 313
  prefix, 312
  suffix, 312
strong orientation, 257
strongly connected components, 257-258
subdivision hierarchy, 447-450
subject of a linked list, 101
substring identifier, 339, 355
suffix tree, 339-352
suffix-minima, 82, 84, 132
suffix-prefix string matching, 362
sum, hypercube, 22
  PRAM, 13-14, 27-31
supervertex, 208
supporting line, 265
switching network, 199
Sylvester matrix, 410-411
symmetry breaking, 75-80
  randomized, 442-443
systolic algorithms, 19-20

tail of binomial distribution, 439
tangent, 265
  upper common, 267
text analysis for string matching, 319-327
Toeplitz matrix, 389-393, 413-415
topological sorting, 259
total order, 61
tour, 258
trace of a matrix, 405
trail in a graph, 256
transitive closure, 246, 249-250
tree computations, 114-118, 126-128
tree contraction, 118-128
tree isomorphism, 142
trees,
  centroid, 141
  inorder traversal, 128, 139, 196
  level, 116
  number of descendants, 118
  postorder traversal, 115-116
  rooting, 114-115
trie, 340
Turan's theorem, 188
Tutte matrix, 475
Tutte theorem, 477-478
two-dimensional array matching, 457-459

uniform probability distribution, 435-436
unique cross-over property, 175

Vandermonde matrix, 410, 428
variance, 437
verification of matrix products, 463-464
vertex cover, 140
vertex of convex hull, 265
visibility polygon, 285-286
visible, 285

Waksman's permutation network, 199
walk in a graph, 258
witness array, 315-317, 327-337
witness function, 315
work, 27, 31-32
work-time (WT) optimal, 32
work-time framework, 27-32
WT scheduling principle, 28

zero counting problem, 509
zero-one principle, 198