An Introduction to Parallel Algorithms

Joseph JáJá
UNIVERSITY OF MARYLAND

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

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:

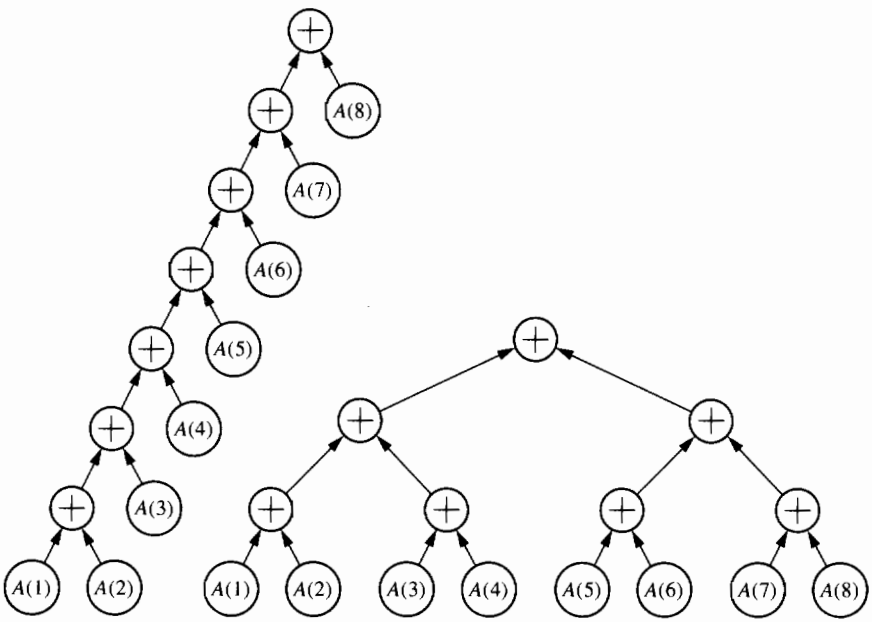

FIGURE 1.1
The dags of two possible algorithms for Example 1.1. (a) A dag for computing the sum of eight elements. (b) An alternate dag for computing the sum of eight elements based on a balanced binary tree scheme.

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

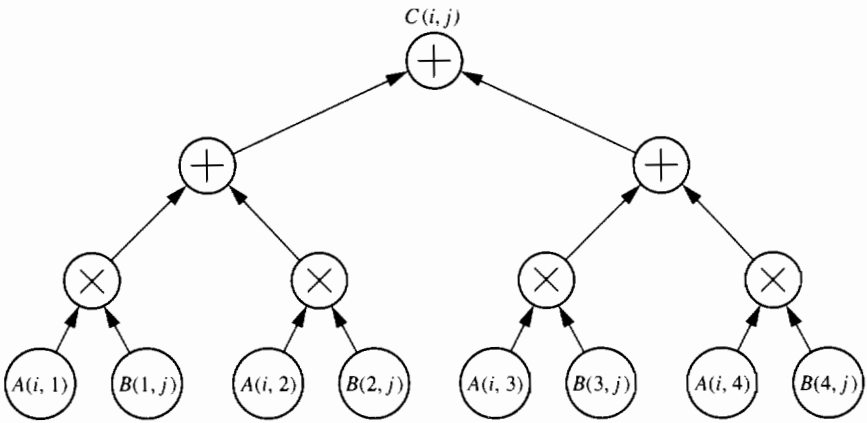

FIGURE 1.2
A dag for computing an entry $C(i, j)$ of the matrix product $C = AB$ for the case of 4×4 matrices.

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.**

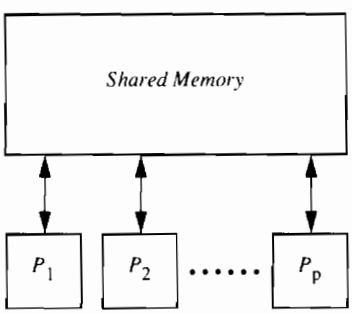

FIGURE 1.3
The shared-memory model.

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 requ by all the processors at step 1. However, no two processors attempt to v into the same location of the shared memory.

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

An important feature of Algorithm 1.1 is that the processors d need to synchronize their activities, given the way the matrix vector pr was partitioned. On the other hand, we can design a parallel algorithm on partitioning A and x into p blocks such that $A = (A_1, A_2, \ldots, A_p)$ a $(x_1, x_2, \ldots, x_p)$, where each A_i is of size $n \times r$ and each x_i is of size product $y = Ax$ is now given by $y = A_1x_1 + A_2x_2 + \cdots + A_rx_r$. processor P_i can compute $z_i = A_ix_i$ after reading A_i and x_i from the memory, for $1 \leq i \leq p$. At this point, however, no processor should be computation of the sum $z_1 + z_2 + \cdots + z_r$ before ensuring that processors have completed their matrix vector products. Therefore plicit synchronization primitive must be placed in each processor's p after the computation of $z_i = A_ix_i$ to force all the processors to sync 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**

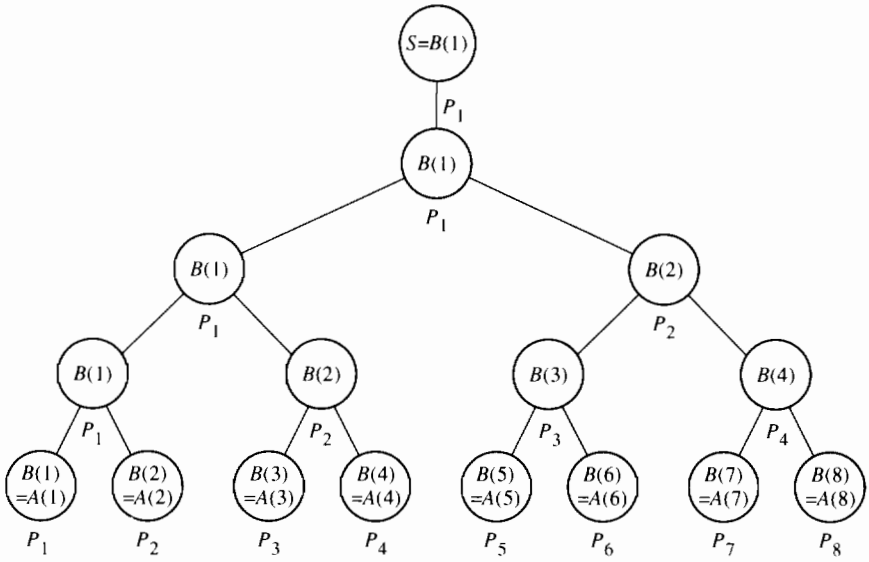

FIGURE 1.4
Computation of the sum of eight elements on a PRAM with eight processors. Each internal node represents a sum operation. The specific processor executing the operation is indicated below each node.

(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 $z_1 + z_2 + \cdots + z_p$.

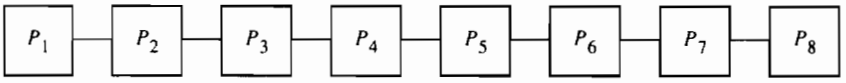

FIGURE 1.5
A linear array of eight processors.

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.

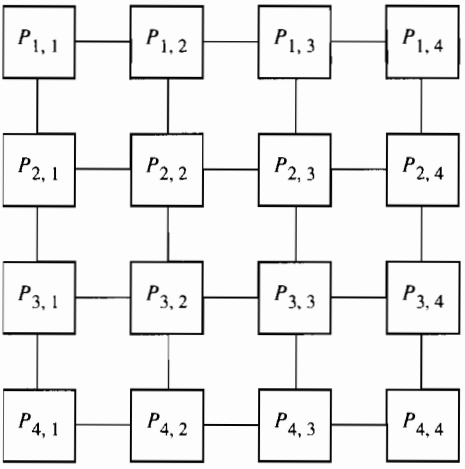

FIGURE 1.6
A 4×4 mesh.

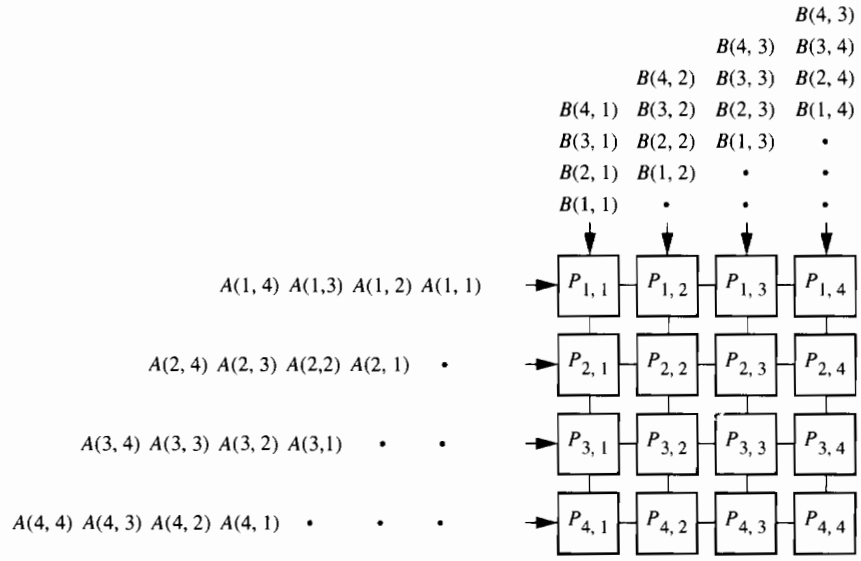

FIGURE 1.7
Matrix multiplication on the mesh using a systolic algorithm. Rows of A move synchronously into the left side, while columns of B move synchronously at the same rate into the top side. When $A(i, l)$ and $B(l, j)$ are available at processor $P_{i,j}$, the operation $C(i, j) = C(i, j) + A(i,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

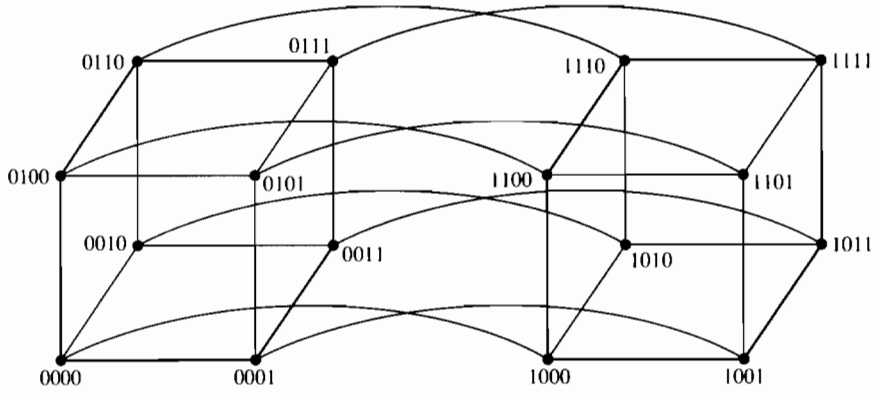

FIGURE 1.8
A four-dimensional hypercube, where the index of each processor is given in binary. Two processors are connected if and only if their indices differ in exactly one bit position.

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 ≤ i ≤ 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.

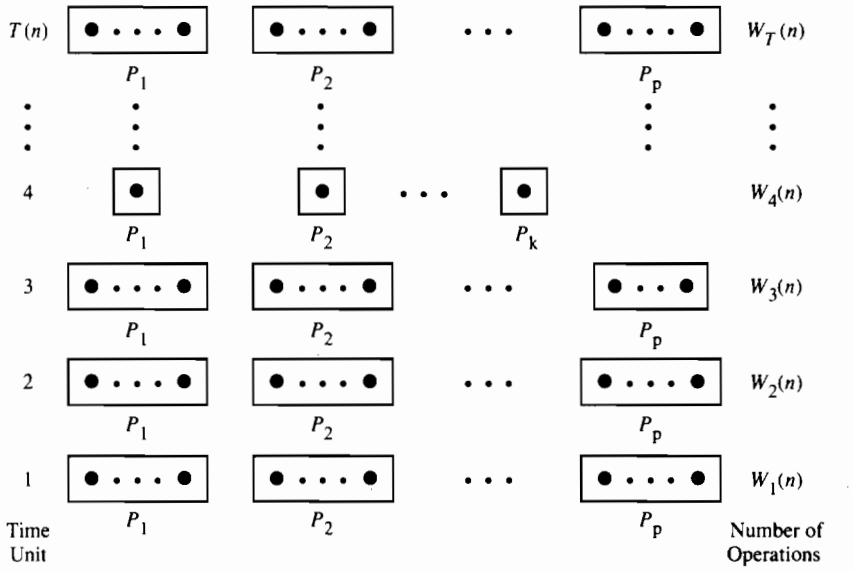

FIGURE 1.9
The WT scheduling principle. During each time unit i, the $W_i(n)$ operations are scheduled as evenly as possible among the available p processors. For example, during time units 1 and 2, each processor is scheduled to execute the same number of operations; during time unit 3, the pth processor 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

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.

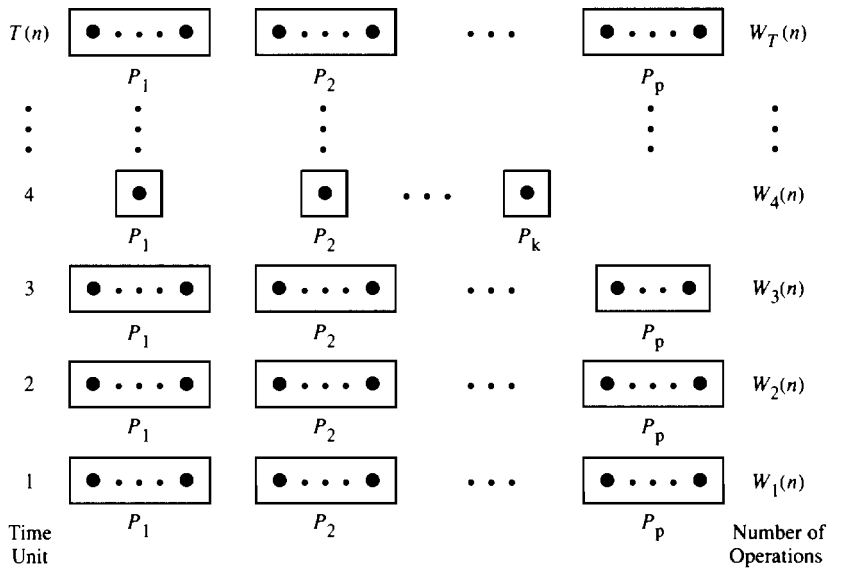

FIGURE 1.9
The WT scheduling principle. During each time unit i, the $W_i(n)$ operations are scheduled as evenly as possible among the available p processors. For example, during time units 1 and 2, each processor is scheduled to execute the same number of operations; during time unit 3, the pth processor 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**

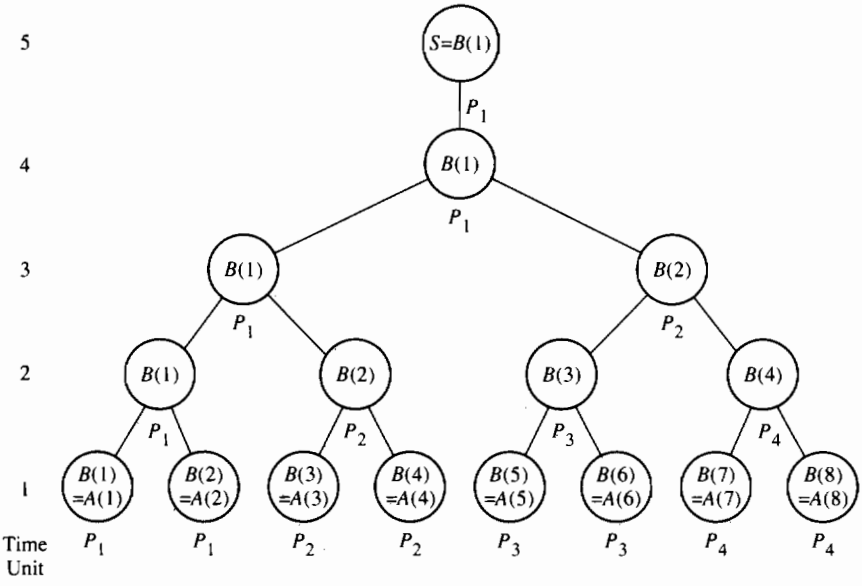

FIGURE 1.10
Processor allocation for computing the sum of eight elements on the PRAM. The operation represented by a node is executed by the processor indicated below the node.

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 $p = O\left(\frac{n}{\log n}\right)$. 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.