CHAPTER 7

Parallel Computation

Parallelism takes many forms and appears in many guises. It is exhibited at the CPU level when microinstructions are executed simultaneously. It is also present when an arithmetic or logic operation is realized by a circuit of small depth, as with carry-save addition. And it is present when multiple computers are connected together in a network. Parallelism can be available but go unused, either because an application was not designed to exploit parallelism or because a problem is inherently serial.

In this chapter we examine a number of explicitly parallel models of computation, including shared and distributed memory models and, in particular, linear and multidimensional arrays, hypercube-based machines, and the PRAM model. We give a broad introduction to a large and representative set of models, describing a handful of good parallel programming techniques and showing through analysis the limits on parallelization. Because of the limited use so far of parallel algorithms and machines, the wide range of hardware and software models developed by the research community has not yet been fully digested by the computer industry.

Parallelism in logic and algebraic circuits is also examined in Chapters 2 and 6. The block I/O model, which characterizes parallelism at the disk level, is presented in Section 11.6 and the classification of problems by their execution time on parallel machines is discussed in Section 8.15.2.

7.1 Parallel Computational Models

A **parallel computer** is any computer that can perform more than one operation at time. By this definition almost every computer is a parallel computer. For example, in the pursuit of speed, computer architects regularly perform multiple operations in each CPU cycle: they execute several microinstructions per cycle and overlap input and output operations (**I/O**) (see Chapter 11) with arithmetic and logical operations. Architects also design parallel computers that are either several CPU and memory units attached to a common bus or a collection of computers connected together via a network. Clearly parallelism is common in computer science today.

However, several decades of research have shown that exploiting large-scale parallelism is very hard. Standard algorithmic techniques and their corresponding data structures do not parallelize well, necessitating the development of new methods. In addition, when parallelism is sought through the undisciplined coordination of a large number of tasks, the sheer number of simultaneous activities to which one human mind must attend can be so large that it is often difficult to insure correctness of a program design. The problems of parallelism are indeed daunting.

Small illustrations of this point are seen in Section 2.7.1, which presents an $O(\log n)$-step, $O(n)$-gate addition circuit that is considerably more complex than the ripple adder given in Section 2.7. Similarly, the fast matrix inversion straight-line algorithm of Section 6.5.5 is more complex than other such algorithms (see Section 6.5).

In this chapter we examine forms of parallelism that are more coarse-grained than is typically found in circuits. We assume that a parallel computer consists of multiple processors and memories but that each processor is primarily serial. That is, although a processor may realize its instructions with parallel circuits, it typically executes only one or a small number of instructions simultaneously. Thus, most of the parallelism exhibited by our parallel computer is due to parallel execution by its processors.

We also describe a few programming styles that encourage a parallel style of programming and offer promise for user acceptance. Finally, we present various methods of analysis that have proven useful in either determining the parallel time needed for a problem or classifying a problem according to its need for parallel time.

Given the doubling of CPU speed every two or three years, one may ask whether we can't just wait until CPU performance catches up with demand. Unfortunately, the appetite for speed grows faster than increases in CPU speed alone can meet. Today many problems, especially those involving simulation of physical systems, require teraflop computers (those performing 10^{12} floating-point operations per second (**FLOPS**)) but it is predicted that petaflop computers (performing 10^{15} FLOPS) are needed. Achieving such high levels of performance with a handful of CPUs may require CPU performance beyond what is physically possible at reasonable prices.

7.2 Memoryless Parallel Computers

The circuit is the premier parallel memoryless computational model: input data passes through a circuit from inputs to outputs and disappears. A circuit is described by a directed acyclic graph in which vertices are either input or computational vertices. Input values and the results of computations are drawn from a set associated with the circuit. (In the case of logic

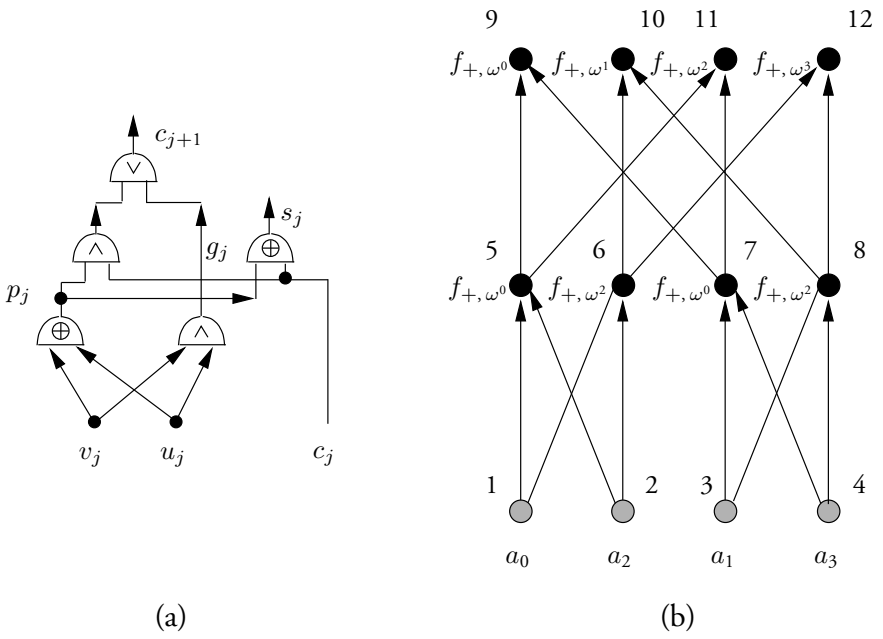

Figure 7.1 Examples of Boolean and algebraic circuits.

circuits, these values are drawn from the set $\mathcal{B} = \{0, 1\}$ and are called Boolean.) The function computed at a vertex is defined through functional composition with values associated with computational and input vertices on which the vertex depends. Boolean logic circuits are discussed at length in Chapters 2 and 9. Algebraic and combinatorial circuits are the subject of Chapter 6. (See Fig. 7.1.)

A circuit is a form of **unstructured parallel computer**. No order or structure is assumed on the operations that are performed. (Of course, this does not prevent structure from being imposed on a circuit.) Generally circuits are a form of **fine-grained parallel computer**; that is, they typically perform low-level operations, such as AND, OR, or NOT in the case of logic circuits, or addition and multiplication in the case of algebraic circuits. However, if the set of values on which circuits operate is rich, the corresponding operations can be complex and coarse-grained.

The **dataflow computer** is a parallel computer designed to simulate a circuit computation. It maintains a list of operations and, when all operands of an operation have been computed, places that operation on a queue of runnable jobs.

We now examine a variety of structured computational models, most of which are coarse-grained and synchronous.

7.3 Parallel Computers with Memory

Many coarse-grained, structured parallel computational models have been developed. In this section we introduce these models as well as a variety of performance measures for parallel computers.

There are many ways to characterize parallel computers. A **fine-grained parallel computer** is one in which the focus is on its constituent components, which themselves consist of low-level entities such as logic gates and binary memory cells. A **coarse-grained parallel computer** is one in which we ignore the low-level components of the computer and focus instead on its functioning at a high level. A complex circuit, such as a carry-lookahead adder, whose details are ignored is a single coarse-grained unit, whereas one whose details are studied explicitly is fine-grained. CPUs and large memory units are generally viewed as coarse-grained.

A parallel computer is a collection of interconnected processors (CPUs or memories). The processors and the media used to connect them constitute a **network**. If the processors are in close physical proximity and can communicate quickly, we often say that they are **tightly coupled** and call the machine a **parallel computer** rather than a computer network. However, when the processors are not in close proximity or when their operating systems require a large amount of time to exchange messages, we say that they are **loosely coupled** and call the machine a **computer network**.

Unless a problem is trivially parallel, it must be possible to exchange messages between processors. A variety of low-level mechanisms are generally available for this purpose. The use of software for the exchange of potentially long messages is called **message passing**. In a tightly coupled parallel computer, messages are prepared, sent, and received quickly relative to the clock speed of its processors, but in a loosely coupled parallel computer, the time required for these steps is much larger. The time T_m to transmit a message from one processor to another is generally assumed to be of the form $T_m = \alpha + l\beta$, where l is the length of the message in words, α (**latency**) is the time to set up a communication channel, and β (**bandwidth**) is the time to send and receive one word. Both α and β are constant multiples of the duration of the CPU clock cycle of the processors. Thus, $\alpha + \beta$ is the time to prepare, send, and receive a single-word message. In a tightly coupled machine α and β are small, whereas in a loosely coupled machine α is large.

An important classification of parallel computers with memory is based on the degree to which they share access to memory. A **shared-memory computer** is characterized by a model in which each processor can address locations in a common memory. (See Fig. 7.2(a).) In this model it is generally assumed that the time to make one access to the common mem-

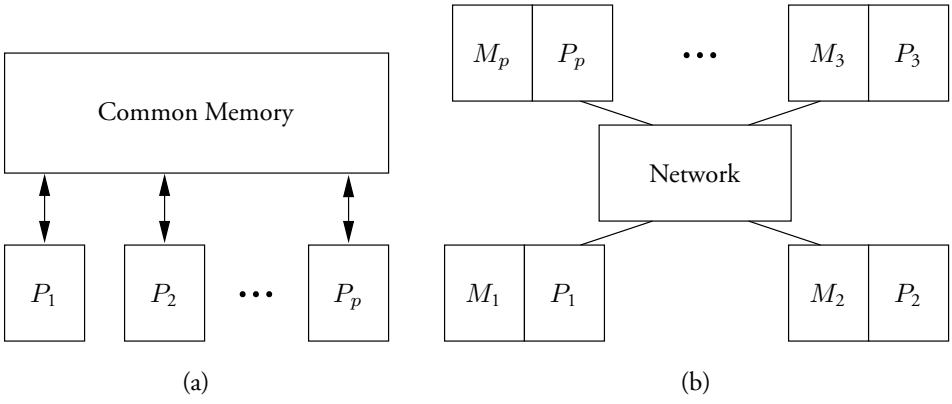

Figure 7.2 (a) A shared-memory computer; (b) a distributed-memory computer.

ory is relatively close to the time for a processor to access one of its registers. Processors in a shared-memory computer can communicate with one another via the common memory. The **distributed-memory computer** is characterized by a model in which processors can communicate with other processors only by sending messages. (See Fig. 7.2(b).) In this model it is generally assumed that processors also have local memories and that the time to send a message from one processor to another can be large relative to the time to access a local memory. A third type of computer, a cross between the first two, is the **distributed shared-memory computer**. It is realized on a distributed-memory computer on which the time to process messages is large relative to the time to access a local memory, but a layer of software gives the programmer the illusion of a shared-memory computer. Such a model is useful when programs can be executed primarily from local memories and only occasionally must access remote memories.

Parallel computers are **synchronous** if all processors perform operations in lockstep and **asynchronous** otherwise. A synchronous parallel machine may alternate between executing instructions and reading from local or common memory. (See the PRAM model of Section 7.9, which is a synchronous, shared-memory model.) Although a synchronous parallel computational model is useful in conveying concepts, in many situations, as with loosely coupled distributed computers, it is not a realistic one. In other situations, such as in the design of VLSI chips, it is realistic. (See, for example, the discussion of systolic arrays in Section 7.5.)

7.3.1 Flynn's Taxonomy

Flynn's taxonomy of parallel computers distinguishes between four extreme types of parallel machine on the basis of the degree of simultaneity in their handling of instructions and data. The **single-instruction, single-data (SISD)** model is a serial machine that executes one instruction per unit time on one data item. An SISD machine is the simplest form of serial computer. The **single-instruction, multiple-data (SIMD)** model is a synchronous parallel machine in which all processors that are not idle execute the same instruction on potentially different data. (See Fig. 7.3.) The **multiple-instruction, single-data (MISD)** model describes a synchronous parallel machine that performs different computations on the same data. While not yet practical, the MISD machine could be used to test the primality of an integer (the single datum) by having processors divide it by independent sets of integers. The

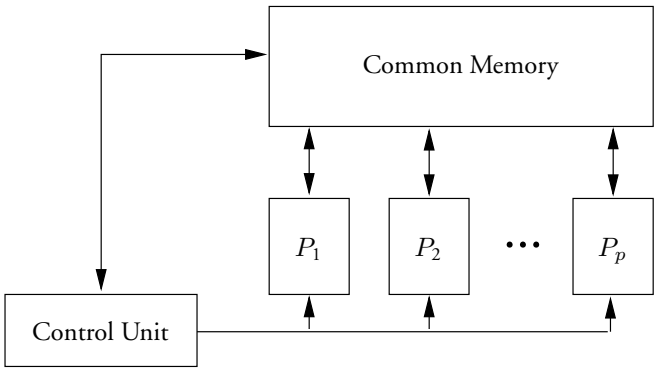

Figure 7.3 In the SIMD model the same instruction is executed on every processor that is not idle.

multiple-instruction, multiple-data (MIMD) model describes a parallel machine that runs a potentially different program on potentially different data on each processor but can send messages among processors.

The SIMD machine is generally designed to have a single instruction decoder unit that controls the action of each processor, as suggested in Fig. 7.3. SIMD machines have not been a commercial success because they require specialized processors rather than today's commodity processors that benefit from economies of scale. As a result, most parallel machines today are MIMD. Nonetheless, the SIMD style of programming remains appealing because programs having a single thread of control are much easier to code and debug. In addition, a MIMD model, the more common parallel model in use today, can be programmed in a SIMD style.

While the MIMD model is often assumed to be much more powerful than the SIMD one, we now show that the former can be converted to the latter with at most a constant factor slowdown in execution time. Let K be the maximum number of different instructions executable by a MIMD machine and index them with integers in the set $\{1, 2, 3, \ldots, K\}$. Slow down the computation of each machine by a factor K as follows: 1) identify time intervals of length K, 2) on the kth step of the jth interval, execute the kth instruction of a processor if this is the instruction that it would have performed on the jth step of the original computation. Otherwise, let the processor be idle by executing its NOOP instruction. This construction executes the instructions of a MIMD computation in a SIMD fashion (all processors either are idle or execute the instruction with the same index) with a slowdown by a factor K in execution time.

Although for most machines this simulation is impractical, it does demonstrate that in the best case a SIMD program is at worst a constant factor slower than the corresponding MIMD program for the same problem. It offers hope that the much simpler SIMD programming style can be made close in performance to the more difficult MIMD style.

7.3.2 The Data-Parallel Model

The **data-parallel model** captures the essential features of the SIMD style. It has a single thread of control in which serial and parallel operations are intermixed. The parallel operations possible typically include vector and shifting operations (see Section 2.5.1), prefix and segmented prefix computations (see Sections 2.6), and data-movement operations such as are realized by a permutation network (see Section 7.8.1). They also include **conditional vector operations**, vector operations that are performed on those vector components for which the corresponding component of an auxiliary flag vector has value 1 (others have value 0).

Figure 7.4 shows a data-parallel program for **radix sort**. This program sorts n d-bit integers, $\{x[n], \ldots, x[1]\}$, represented in binary. The program makes d passes over the integers. On each pass the program reorders the integers, placing those whose jth least significant bit (**lsb**) is 1 ahead of those for which it is 0. This reordering is **stable**; that is, the previous ordering among integers with the same jth lsb is retained. After the jth pass, the n integers are sorted according to their j least significant bits, so that after d passes the list is fully sorted. The prefix function $\mathcal{P}_+^{(n)}$ computes the running sum of the jth lsb on the jth pass. Thus, for k such that $x[k]_j = 1$ (0), b_k (c_k) is the number of integers with index k or higher whose jth lsb is 1 (0). The value of $a_k = b_k x[k]_j + (c_k + b_1)\overline{x[k]_j}$ is b_k or $c_k + b_1$, depending on whether the lsb of $x[k]$ is 1 or 0, respectively. That is, a_k is the index of the location in which the kth integer is placed after the jth pass.

{ $x[n]_j$ is the jth least significant bit of the nth integer. }
{ After the jth pass, the integers are sorted by their j least significant bits. }
{ Upon completion, the kth location contains the kth largest integer. }

for j := 0 **to** $d-1$
begin
$(b_n, \ldots, b_1) := \mathcal{P}_+^{(n)}(x[n]_j, \ldots, x[1]_j)$;
{ b_k is the number of 1's among $x[n]_j, \ldots, x[k]_j$. }
{ b_1 is the number of integers whose jth bit is 1. }

$(c_n, \ldots, c_1) := \mathcal{P}_+^{(n)}(\overline{x[n]_j}, \ldots, \overline{x[1]_j})$;
{ c_k is the number of 0's among $x[n]_j, \ldots, x[k]_j$. }

$(a_n, \ldots, a_1) := \left(b_n x[n]_j + (c_n + b_1)\overline{x[n]_j}, \ldots, b_1 x[1]_j + (c_1 + b_1)\overline{x[1]_j}\right)$;
{ $a_k = b_k x[k]_j + (c_k + b_1)\overline{x[k]_j}$ is the rank of the kth key. }

$(x[n+1-a_n], x[n+1-a_{n-1}], \ldots, x[n+1-a_1]) := (x[n], x[n-1], \ldots, x[1])$
{ This operation permutes the integers. }
end

Figure 7.4 A data-parallel radix sorting program to sort n d-bit binary integers that makes two uses of the prefix function $\mathcal{P}_+^{(n)}$.

The data-parallel model is often implemented using the **single-program multiple-data (SPMD)** model. This model allows copies of one program to run on multiple processors with potentially different data without requiring that the copies run in synchrony. It also allows the copies to synchronize themselves periodically for the transfer of data. A convenient abstraction often used in the data-parallel model that translates nicely to the SPMD model is the assumption that a collection of virtual processors is available, one per vector component. An operating system then maps these virtual processors to physical ones. This method is effective when there are many more virtual processors than real ones so that the time for interprocessor communication is amortized.

7.3.3 Networked Computers

A **networked computer** consists of a collection of processors with direct connections between them. In this context a processor is a CPU with memory or a sequential machine designed to route messages between processors. The graph of a network has a vertex associated with each processor and an edge between two connected processors. Properties of the graph of a network, such as its **size** (number of vertices), its **diameter** (the largest number of edges on the shortest path between two vertices), and its **bisection width** (the smallest number of edges between a subgraph and its complement, both of which have about the same size) characterize its computational performance. Since a transmission over an edge of a network introduces delay, the diameter of a network graph is a crude measure of the worst-case time to transmit

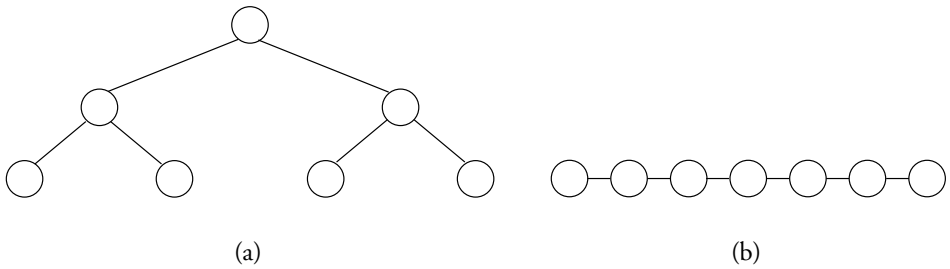

Figure 7.5 Completely balanced (a) and unbalanced (b) trees.

a message between processors. Its bisection width is a measure of the amount of information that must be transmitted in the network for processors to communicate with their neighbors.

A large variety of networks have been investigated. The graph of a **tree network** is a tree. Many simple tasks, such as computing sums and broadcasting (sending a message from one processor to all other processors), can be done on tree networks. Trees are also naturally suited to many recursive computations that are characterized by **divide-and-conquer strategies**, in which a problem is divided into a number of like problems of similar size to yield small results that can be combined to produce a solution to the original problem. Trees can be completely balanced or unbalanced. (See Fig. 7.5.) Balanced trees of fixed degree have a root and bounded number of edges associated with each vertex. The diameter of such trees is logarithmic in the number of vertices. Unbalanced trees can have a diameter that is linear in the number of vertices.

A mesh is a regular graph (see Section 7.5) in which each vertex has the same degree except possibly for vertices on its boundary. Meshes are well suited to matrix operations and can be used for a large variety of other problems as well. If, as some believe, speed-of-light limitations will be an important consideration in constructing fast computers in the future [43], the one-, two-, and three-dimensional mesh may very well become the computer organization of choice. The diameter of a mesh of dimension d with n vertices is proportional to $n^{1/d}$. It is not as small as the diameter of a tree but acceptable for tasks for which the cost of communication can be amortized over the cost of computation.

The hypercube (see Section 7.6) is a graph that has one vertex at each corner of a multidimensional cube. It is an important conceptual model because it has low (logarithmic) diameter, large bisection width, and a connectivity for which it is easy to construct efficient parallel algorithms for a large variety of problems. While the hypercube and the tree have similar diameters, the superior connectivity of the hypercube leads to algorithms whose running time is generally smaller than on trees. Fortunately, many hypercube-based algorithms can be efficiently translated into algorithms for other network graphs, such as meshes.

We demonstrate the utility of each of the above models by providing algorithms that are naturally suited to them. For example, linear arrays are good at performing matrix-vector multiplications and sorting with bubble sort. Two-dimensional meshes are good at matrix-matrix multiplication, and can also be used to sort in much less time than linear arrays. The hypercube network is very good at solving a variety of problems quickly but is much more expensive to realize than linear or two-dimensional meshes because each processor is connected to many more other processors.

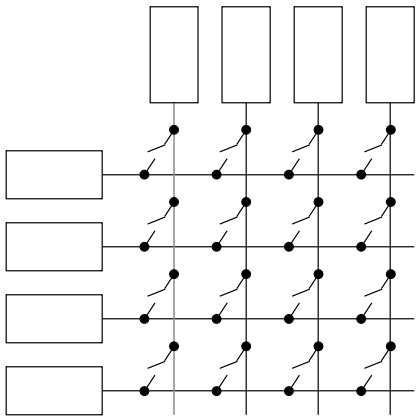

Figure 7.6 A crossbar connection network. Any two processors can be connected.

In designing parallel algorithms it is often helpful to devise an algorithm for a particular parallel machine model, such as a hypercube, and then map the hypercube and the algorithm with it to the model of the machine on which it will be executed. In doing this, the question arises of how efficiently one graph can be embedded into another. This is the **graph-embedding** problem. We provide an introduction to this important question by discussing embeddings of one type of machine into another.

A **connection network** is a network computer in which all vertices except for peripheral vertices are used to route messages. The peripheral vertices are the computers that are connected by the network. One of the simplest such networks is the **crossbar network**, in which a row of processors is connected to a column of processors via a two-dimensional array of switches. (See Fig. 7.6.) The crossbar switch with $2n$ computational processors has n^2 routing vertices. The butterfly network (see Fig. 7.15) provides a connectivity similar to that of the crossbar but with many fewer routing vertices. However, not all permutations of the inputs to a butterfly can be mapped to its outputs. For this purpose the Beneš network (see Fig. 7.20) is better suited. It consists of two butterfly graphs with the outputs of one graph connected to the outputs of the second and the order of edges of the second reversed. Many other permutation networks exist. Designers of connection networks are very concerned with the variety of connections that can be made among computational processors, the time to make these connections, and the number of vertices in the network for the given number of computational processors. (See Section 7.8.)

7.4 The Performance of Parallel Algorithms

We now examine measures of performance of parallel algorithms. Of these, computation time is the most important. Since parallel computation time T_p is a function of p, the number of processors used for a computation, we seek a relationship among p, T_p, and other measures of the complexity of a problem.

Given a p-processor parallel machine that executes T_p steps, in the spirit of Chapter 3, we can construct a circuit to simulate it. Its size is proportional to pT_p, which plays the role of

serial time T_s. Similarly, a single-processor RAM of the type used in a p-processor parallel machine but with p times as much memory can simulate an algorithm on the parallel machine in p times as many steps; it simulates each step of each of the p RAM processors in succession. This observation provides the following relationship among p, T_p, and T_s when storage space for the serial and parallel computations is comparable.

THEOREM 7.4.1 *Let T_s be the smallest number of steps needed on a single RAM with storage capacity S, in bits, to compute a function f. If f can be computed in T_p steps on a network of p RAM processors, each with storage S/p, then T_p satisfies the following inequality:*

$$pT_p \geq T_s \tag{7.1}$$

Proof This result follows because, while the serial RAM can simulate the parallel machine in pT_p steps, it may be able to compute the function in question more quickly. ■

The **speedup** $\mathcal{S}$ of a parallel p-processor algorithm over the best serial algorithm for a problem is defined as $\mathcal{S} = T_s/T_p$. We see that, with p processors, a speedup of at most p is possible; that is, $\mathcal{S} \leq p$. This result can also be stated in terms of the computational work done by serial and parallel machines, defined as the number of equivalent serial operations. (Computational work is defined in terms of the equivalent number of gate operations in Section 3.1.2. The two measures differ only in terms of the units in which work is measured, CPU operations in this section and gate operations in Section 3.1.2.) The **computational work** W_p done by an algorithm on a p-processor RAM machine is $W_p = pT_p$. The above theorem says that the minimal parallel work needed to compute a function is at least the serial work required for it, that is, $W_p \geq W_s = T_s$. (Note that we compare the work on a serial processor to a collection of p identical processors, so that we need not take into account differences among processors.)

A **parallel algorithm is efficient** if the work that it does is close to the work done by the best serial algorithm. A **parallel algorithm is fast** if it achieves a nearly maximal speedup. We leave unspecified just how close to optimal a parallel algorithm must be for it to be classified as efficient or fast. This will often be determined by context. We observe that parallel algorithms may be useful if they complete a task with acceptable losses in efficiency or speed, even if they are not optimal by either measure.

7.4.1 Amdahl's Law

As a warning that it is not always possible with p processors to obtain a speedup of p, we introduce Amdahl's Law, which provides an intuitive justification for the difficulty of parallelizing some tasks. In Sections 3.9 and 8.9 we provide concrete information on the difficulty of parallelizing individual problems by introducing the **P**-complete problems, problems that are the hardest polynomial-time problems to parallelize.

THEOREM 7.4.2 (Amdahl's Law) *Let f be the fraction of a program's execution time on a serial RAM that is parallelizable. Then the speedup S achievable by this program on a p-processor RAM machine must satisfy the following bound:*

$$S \leq \frac{1}{(1-f) + f/p}$$

Proof Given a T_s-step serial computation, fT_s/p is the smallest possible number of steps on a p-processor machine for the parallelizable serial steps. The remaining $(1-f)T_s$ serial

steps take at least the same number of steps on the parallel machine. Thus, the parallel time T_p satisfies $T_p \geq T_s[(1-f)+f/p]$ from which the result follows. ■

This result shows that if a fixed fraction f of a program's serial execution time can be parallelized, the speedup achievable with that program on a parallel machine is bounded above by $1/(1-f)$ as p grows without limit. For example, if 90% of the time of a serial program can be parallelized, the maximal achievable speed is 10, regardless of the number of parallel processors available.

While this statement seems to explain the difficulty of parallelizing certain algorithms, it should be noted that programs for serial and parallel machines are generally very different. Thus, it is not reasonable to expect that analysis of a serial program should lead to bounds on the running time of a parallel program for the same problem.

7.4.2 Brent's Principle

We now describe how to convert the inherent parallelism of a problem into an efficient parallel algorithm. Brent's principle, stated in Theorem 7.4.3, provides a general schema for exploiting parallelism in a problem.

THEOREM 7.4.3 *Consider a computation C that can be done in t parallel steps when the time to communicate between operations can be ignored. Let m_i be the number of primitive operations done on the ith step and let $m = \sum_{i=1}^{t} m_i$. Consider a p-processor machine M capable of the same primitive operations, where $p \leq \max_i m_i$. If the communication time between the operations in C on M can be ignored, the same computation can be performed in T_p steps on M, where T_p satisfies the following bound:*

$$T_p \leq (m/p) + t$$

Proof A parallel step in which m_i operations are performed can be simulated by M in $\lceil m_i/p \rceil < (m_i/p) + 1$ steps, from which the result follows. ■

Brent's principle provides a schema for realizing the inherent parallelism in a problem. However, it is important to note that the time for communication between operations can be a serious impediment to the efficient implementation of a problem on a parallel machine. Often, the time to route messages between operations can be the most important limitation on exploitation of parallelism.

We illustrate Brent's principle with the problem of adding n integers, $x_1, \ldots, x_n$, $n = 2^k$. Under the assumption that at most two integers can be added in one primitive operation, we see that the sum can be formed by performing $n/2$ additions, $n/4$ additions of these results, etc., until the last sum is formed. Thus, $m_i = n/2^i$ for $i \leq \lceil \log_2 n \rceil$. When only p processors are available, we assign $\lceil n/p \rceil$ integers to $p-1$ processors and $n-(p-1)\lceil n/p \rceil$ integers to the remaining processor. In $\lceil n/p \rceil$ steps, the p processors each compute their local sums, leaving their results in a reserved location. In each subsequent phase, half of the processors active in the preceding phase are active in this one. Each active processor fetches the partial sum computed by one other processor, adds it to its partial sum, and stores the result in a reserved place. After $O(\log p)$ phases, the sum of the n integers has been computed. This algorithm computes the sum of the n integers in $O(n/p + \log p)$ time steps. Since the maximal speedup possible is p, this algorithm is optimal to within a constant multiplicative factor if $\log p \leq (n/p)$ or $p \leq n/\log n$.

It is important to note that the time to communicate between processors is often very large relative to the length of a CPU cycle. Thus, the assumption that it takes zero time to communicate between processors, the basis of Brent's principle, holds only for tightly coupled processors.

7.5 Multidimensional Meshes

In this section we examine multidimensional meshes. A **one-dimensional mesh** or **linear array** of processors is a one-dimensional (1D) array of computing elements connected via nearest-neighbor connections. (See Fig. 7.7.) If the vertices of the array are indexed with integers from the set $\{1, 2, 3, \ldots, n\}$, then vertex i, $2 \leq i \leq n-1$, is connected to vertices $i-1$ and $i+1$. If the linear array is a **ring**, vertices 1 and n are also connected. Such an end-to-end connection can be made with short connections by folding the linear array about its midpoint.

The linear array is an important model that finds application in very large-scale integrated (VLSI) circuits. When the processors of a linear array operate in synchrony (which is the usual way in which they are used), it is called a linear **systolic array** (a systole is a recurrent rhythmic contraction, especially of the heart muscle). A systolic array is any mesh (typically 1D or 2D) in which the processors operate in synchrony. The computing elements of a systolic array are called **cells**. A linear systolic array that convolves two binary sequences is described in Section 1.6.

A **multidimensional mesh** (see Fig. 7.8) (or **mesh**) offers better connectivity between processors than a linear array. As a consequence, a multidimensional mesh generally can compute functions more quickly than a 1D one. We illustrate this point by matrix multiplication on 2D meshes in Section 7.5.3.

Figure 7.8 shows 2D and 3D meshes. Each vertex of the 2D mesh is numbered by a pair (r, c), where $0 \leq r \leq n-1$ and $0 \leq c \leq n-1$ are its row and column indices. (If the cell

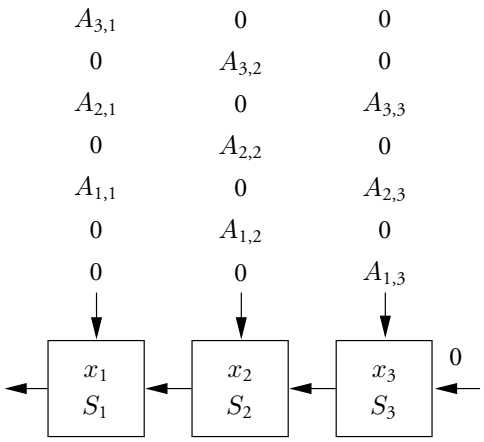

Figure 7.7 A linear array to compute the matrix-vector product $A\boldsymbol{x}$, where $A = [a_{i,j}]$ and $\boldsymbol{x}^T = (x_1, \ldots, x_n)$. On each cycle, the ith processor sets its current sum, S_i, to the sum to its right, S_{i+1}, plus the product of its local value, x_i, with its vertical input.

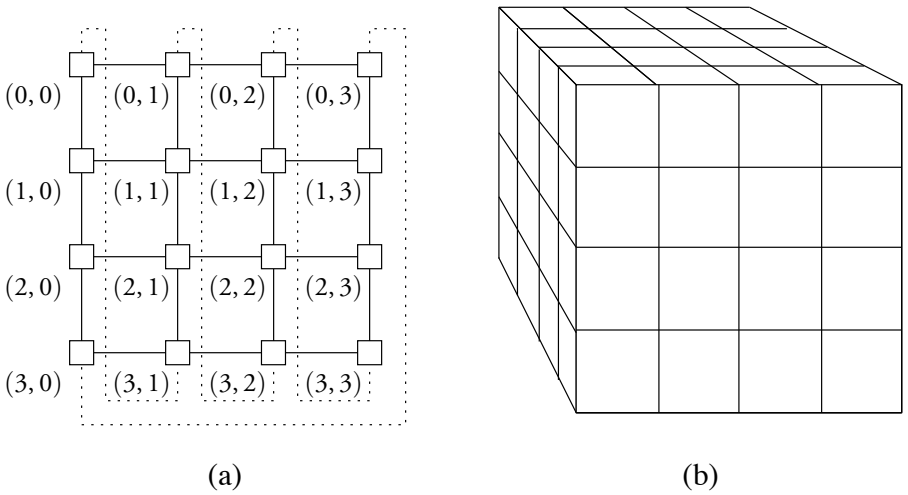

Figure 7.8 (a) A two-dimensional mesh with optional connections between the boundary elements shown by dashed lines. (b) A 3D mesh (a cube) in which elements are shown as subcubes.

(r, c) is associated with the integer $rn + c$, this is the **row-major order** of the cells. Cells are numbered left-to-right from 0 to 3 in the first row, 4 to 7 in the second, 8 to 11 in the third, and 12 to 15 in the fourth.) Vertex (r, c) is adjacent to vertices $(r - 1, c)$ and $(r + 1, c)$ for $1 \leq r \leq n - 2$. Similarly, vertex (r, c) is adjacent to vertices $(r, c - 1)$ and $(r, c + 1)$ for $1 \leq c \leq n - 2$. Vertices on the boundaries may or may not be connected to other boundary vertices, and may be connected in a variety of ways. For example, vertices in the first row (column) can be connected to those in the last row (column) in the same column (row) (this is a **toroidal mesh**) or the next larger column (row). The second type of connection is associated with the dashed lines in Fig. 7.8(a).

Each vertex in a 3D mesh is indexed by a triple (x, y, z), $0 \leq x, y, z \leq n-1$, as suggested in Fig. 7.8(b). Connections between boundary vertices, if any, can be made in a variety of ways. Meshes with larger dimensionality are defined in a similar fashion.

A d-dimensional mesh consists of processors indexed by a d-tuple $(n_1, n_2, \ldots, n_d)$ in which $0 \leq n_j \leq N_j - 1$ for $1 \leq j \leq d$. If processors $(n_1, n_2, \ldots, n_d)$ and $(m_1, m_2, \ldots, m_d)$ are adjacent, there is some j such that $n_i = m_i$ for $j \neq i$ and $|n_j - m_j| = 1$. There may also be connections between boundary processors, that is, processors for which one component of their index has either its minimum or maximum value.

7.5.1 Matrix-Vector Multiplication on a Linear Array

As suggested in Fig. 7.7, the cells in a systolic array can have external as well as nearest-neighbor connections. This systolic array computes the matrix-vector product $A\boldsymbol{x}$ of an $n \times n$ matrix with an n-vector. (In the figure, $n = 3$.) The cells of the systolic array beat in a rhythmic fashion. The ith processor sets its current sum, S_i, to the product of x_i with its vertical input plus the value of S_{i+1} to its right (the value 0 is read by the rightmost cell). Initially, $S_i = 0$ for $1 \leq i \leq n$. Since alternating vertical inputs are 0, the alternating values of S_i are 0. In Fig. 7.7 the successive values of S_3 are $A_{1,3}x_3$, 0, $A_{2,3}x_3$, 0, $A_{3,3}x_3$, 0, 0. The successive values of S_2

are 0, $A_{1,2}x_2 + A_{1,3}x_3$, 0, $A_{2,2}x_2 + A_{2,3}x_3$, 0, $A_{3,2}x_2 + A_{3,3}x_3$, 0. The successive values of S_1 are 0, 0, $A_{1,1}x_1 + A_{1,2}x_2 + A_{1,3}x_3$, 0, $A_{2,1}x_1 + A_{2,2}x_2 + A_{2,3}x_3$, 0, $A_{3,1}x_1 + A_{3,2}x_2 + A_{3,3}x_3$.

The algorithm described above to compute the matrix-vector product for a 3×3 matrix clearly extends to arbitrary $n \times n$ matrices. (See Problem 7.8.) Since the last element of an $n \times n$ matrix arrives at the array after $3n - 2$ time steps, such an array will complete its task in $3n-1$ time steps. A lower bound on the time for this problem (see Problem 7.9) can be derived by showing that the n^2 entries of the matrix A and the n entries of the matrix $\boldsymbol{x}$ must be read to compute $A\boldsymbol{x}$ correctly by an algorithm, whether serial or not. By Theorem 7.4.1 it follows that all systolic algorithms using n processors require n steps. Thus, the above algorithm is nearly optimal to within a constant multiplicative factor.

THEOREM 7.5.1 *There exists a linear systolic array with n cells that computes the product of an $n \times n$ matrix with an n-vector in $3n - 1$ steps, and no algorithm on such an array can do this computation in fewer than n steps.*

Since the product of two $n \times n$ matrices can be realized as n matrix-vector products with an $n \times n$ matrix, an n-processor systolic array exists that can multiply two matrices nearly optimally.

7.5.2 Sorting on Linear Arrays

A second application of linear systolic arrays is bubble sorting of integers. A sequential version of the **bubble sort** algorithm passes over the entries in a tuple $(x_1, x_2, \ldots, x_n)$ from left to right multiple times. On the first pass it finds the largest element and moves it to the rightmost position. It applies the same procedure to the first $n-1$ elements of the resultant list, stopping when it finds a list containing one element. This sequential procedure takes time proportional to $n + (n-1) + (n-2) + \cdots + 2 + 1 = n(n+1)/2$.

A parallel version of bubble sort, sometimes called **odd-even transposition sort**, is naturally realized on a linear systolic array. The n entries of the array are placed in n cells. Let c_i be the word in the ith cell. We assume that in one unit of time two adjacent cells can read words stored in each other's memories (c_i and c_{i+1}), compare them, and swap them if one (c_i) is larger than the other (c_{i+1}). The odd-even transposition sort algorithm executes n stages. In the even-numbered stages, integers in even-numbered cells are compared with integers in the next higher numbered cells and swapped, if larger. In the odd-numbered stages, the same operation is performed on integers in odd-numbered cells. (See Fig. 7.9.) We show that in n steps the sorting is complete.

THEOREM 7.5.2 *Bubble sort of n elements on a linear systolic array can be done in at most n steps. Every algorithm to sort a list of n elements on a linear systolic array requires at least $n - 1$ steps. Thus, bubble sort on a linear systolic array is almost optimal.*

Proof To derive the upper bound we use the zero-one principle (see Theorem 6.8.1), which states that if a comparator network for inputs over an ordered set $\mathcal{A}$ correctly sorts all binary inputs, it correctly sorts all inputs. The bubble sort systolic array maps directly to a comparator network because each of its operations is data-independent, that is, **oblivious**. To see that the systolic array correctly sorts binary sequences, consider the position, r, of the rightmost 1 in the array.

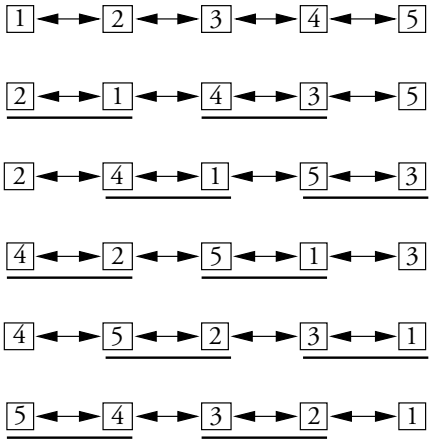

Figure 7.9 A systolic implementation of bubble sort on a sequence of five items. Underlined pairs of items are compared and swapped if out of order. The bottom row shows the first set of comparisons.

If r is even, on the first phase of the algorithm this 1 does not move. However, on all subsequent phases it moves right until it arrives at its final position. If r is odd, it moves right on all phases until it arrives in its final position. Thus by the second step the rightmost 1 moves right on every step until it arrives at its final position. The second rightmost 1 is free to move to the right without being blocked by the first 1 after the second phase. This second 1 will move to the right by the third phase and continue to do so until it arrives at its final position. In general, the kth rightmost 1 starts moving to the right by the $(k+1)$st phase and continues until it arrives at its final position. It follows that at most n phases are needed to sort the 0-1 sequence. By the zero-one principle, the same applies to all sequences.

To derive the lower bound, assume that the sorted elements are increasing from left to right in the linear array. Let the elements initially be placed in decreasing order from left to right. Thus, the process of sorting moves the largest element from the leftmost location in the array to the rightmost. This requires at least $n-1$ steps. The same lower bound holds if some other permutation of the n elements is desired. For example, if the kth largest element resides in the rightmost cell at the end of the computation, it can reside initially in the leftmost cell, requiring at least $n-1$ operations to move to its final position. ■

7.5.3 Matrix Multiplication on a 2D Mesh

2D systolic arrays are natural structures on which to compute the product $C = A \times B$ of matrices A and B. (Matrix multiplication is discussed in Section 6.3.) Since $C = A \times B$ can be realized as n matrix-vector multiplications, C can be computed with n linear arrays. (See Fig. 7.7.) If the columns of B are stored in successive arrays and the entries of A pass from one array to the next in one unit of time, the nth array receives the last entry of B after $4n-2$ time steps. Thus, this 2D systolic array computes $C = A \times B$ in $4n-1$ steps. Somewhat more efficient 2D systolic arrays can be designed. We describe one of them below.

Figure 7.10 shows a 2D mesh for matrix multiplication. Each cell of this mesh adds to its stored value the product of the value arriving from above and to its left. These two values pass through the cells to those below and to their right, respectively. When the entries of A are supplied on the left and those of B are supplied from above in the order shown, the cell $C_{i,j}$ computes $c_{i,j}$, the (i, j) entry of the product matrix C. For example, cell $C_{2,3}$ accumulates the value $c_{2,3} = a_{2,1} * b_{1,3} + a_{2,2} * b_{2,3} + a_{2,3} * b_{3,3}$. After the entries of C have been computed, they are produced as outputs by shifting the entries of the mesh to one side of the array. When generalized to $n \times n$ matrices, this systolic array requires $2n - 1$ steps for the last of the matrix components to enter the array, and another $n - 1$ steps to compute the last entry $c_{n,n}$. An additional n steps are needed to shift the components of the product matrix out of the array. Thus, this systolic array performs matrix multiplication in $4n - 2$ steps.

We put the following requirements on every systolic array (of any dimension) that computes the matrix multiplication function: a) each component of each matrix enters the array at one location, and b) each component of the product matrix is computed at a unique cell. We now show that the systolic matrix multiplication algorithm is optimal to within a constant multiplicative factor.

THEOREM 7.5.3 *Two $n \times n$ matrices can be multiplied by an $n \times n$ systolic array in $4n - 2$ steps and every two-dimensional systolic array for this problem requires at least $(n/2) - 1$ steps.*

Proof The proof that two $n \times n$ matrices can be multiplied in $4n - 2$ steps by a two-dimensional systolic array was given above. We now show that $\Omega(n)$ steps are required to multiply two $n \times n$ matrices, A and B, to produce the matrix $C = A \times B$. Observe that the number of cells in a two-dimensional array that are within d moves from any particular cell is at most $\sigma(d)$, where $\sigma(d) = 2d^2 + 2d + 1$. The maximum occurs at the center of the array. (See Problem 7.11.)

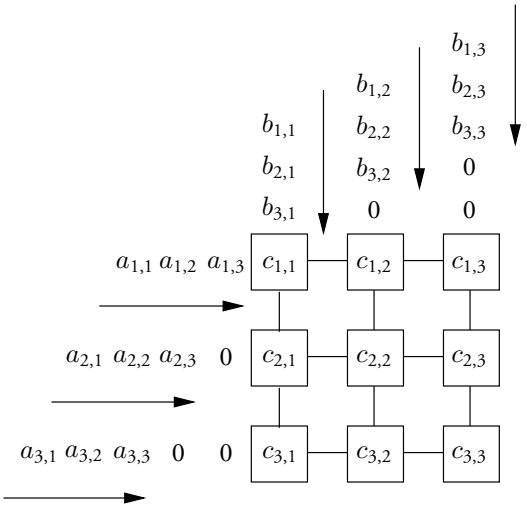

Figure 7.10 A two-dimensional mesh for the multiplication of two matrices. The entries in these matrices are supplied in successive time intervals to processors on the boundary of the mesh.

Given a systolic array with inputs supplied externally over time (see Fig. 7.10), we enlarge the array so that each component of each matrix is initially placed in a unique cell. The enlarged array contains the original $n \times n$ array.

Let $C = [c_{i,j}]$. Because $c_{i,j} = \sum_u a_{i,u} b_{u,j}$, it follows that for each value of i, j, t, and u there is a path from $a_{i,u}$ to the cell at which $c_{i,j}$ is computed as well as a path from $b_{t,j}$ to this same cell. Thus, it follows that there is a path in the array between arbitrary entries $a_{i,u}$ and $b_{t,j}$ of the matrices $A = [a_{i,u}]$ and $B = [b_{t,j}]$. Let s be the maximum number of array edges between an element of A or B and an element of C on which it depends. It follows that at least s steps are needed to form C and that every element of A and B is within distance $2s$. Furthermore, each of the $2n^2$ elements of A and B is located initially in a unique cell of the expanded systolic array. Since there are at most $\sigma(2s)$ vertices within a distance of $2s$, it follows that $\sigma(2s) = 2(2s)^2 + 2(2s) + 1 \geq 2n^2$, from which we conclude that the number of steps to multiply $n \times n$ matrices is at least $s \geq \frac{1}{2}(n^2 - \frac{1}{4})^{1/2} - \frac{1}{4} \geq \frac{n}{2} - 1$. ■

7.5.4 Embedding of 1D Arrays in 2D Meshes

Given an algorithm for a linear array, we ask whether that algorithm can be efficiently realized on a 2D mesh. This is easily determined: we need only specify a mapping of the cells of a linear array to cells in the 2D mesh. Assuming that the two arrays have the same number of cells, a natural mapping is obtained by giving the cells of an $n \times n$ mesh the **snake-row ordering**. (See Fig. 7.11.) In this ordering cells of the first row are ordered from left to right and numbered from 0 to $n - 1$; those in the second row are ordered from right to left and numbered from n to $2n - 1$. This process repeats, alternating between ordering cells from left to right and right to left and numbering the cells in succession. Ordering the cells of a linear array from left to right and numbering them from 0 to $n^2 - 1$ allows us to map the linear array directly to the 2D mesh. Any algorithm for the linear array runs in the same time on a 2D mesh if the processors in the two cases are identical.

Now we ask if, given an algorithm for a 2D mesh, we can execute it on a linear array. The answer is affirmative, although the execution time of the algorithm may be much greater on the 1D array than on the 2D mesh. As a first step, we map vertices of the 2D mesh onto vertices of the 1D array. The snake-row ordering of the cells of an $n \times n$ array provides a convenient

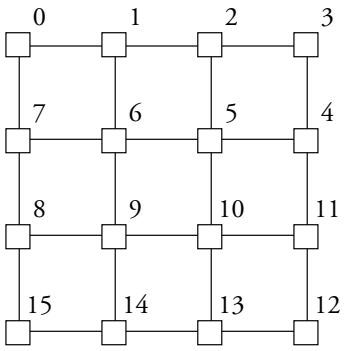

Figure 7.11 Snake-row ordering of the vertices of a two-dimensional mesh.

mapping of the cells of the 2D mesh onto the cells of the linear array with n^2 cells. We assume that each of the cells of the linear array is identical to a cell in the 2D mesh.

We now address the question of communication between cells. When mapped to the 1D array, cells can communicate only with their two immediate neighbors in the array. However, cells on the $n \times n$ mesh can communicate with as many as four neighbors. Unfortunately, cells in one row of the 2D mesh that are neighbors of cells in an adjacent row are mapped to cells that are as far as $2n - 1$ cells away in the linear array. We show that with a factor of $8n - 2$ slowdown, the linear array can simulate the 2D mesh. A slowdown by at least a factor of $n/2$ is necessary for those problems and data for which a datum moves from the first to the last entry in the array (in $n^2 - 1$ steps) to simulate a movement that takes $2n - 1$ steps on the array. ($(n^2 - 1)/(2n - 1) \geq n/2$ for $n \geq 2$.)

Given an algorithm for a 2D mesh, slow it down as follows:

a) Subdivide each cycle into six subcycles.

b) In the first of these subcycles let each cell compute using its local data.

c) In the second subcycle let each cell communicate with neighbor(s) in adjacent columns.

d) In the third subcycle let cells in even-numbered rows send messages to cells in the next higher numbered rows.

e) In the fourth subcycle let cells in even-numbered rows receive messages from cells in the next higher numbered rows.

f) In the fifth subcycle let cells in odd-numbered rows send messages to cells in next higher numbered rows.

g) In the sixth subcycle let cells in odd-numbered rows receive messages from cells in next higher numbered rows.

When the revised 2D algorithm is executed on the linear array, computation occurs in the first subcycle in unit time. During the second subcycle communication occurs in unit time because cells that are column neighbors in the 2D mesh are adjacent in the 1D array. The remaining four subcycles involve communication between pairs of groups of n cells each. This can be done for all pairs in $2n - 1$ time steps: each cell shifts a datum in the direction of the cell for which it is destined. After $2n - 1$ steps it arrives and can be processed. We summarize this result below.

THEOREM 7.5.4 *Any T-step systolic algorithm on an $n \times n$ array can be simulated on a linear systolic array with n^2 cells in at most $(8n - 2)T$ steps.*

In the next section we demonstrate that hypercubes can be embedded into meshes. From this result we derive mesh-based algorithms for a variety of problems from hypercube-based algorithms for these problems.

7.6 Hypercube-Based Machines

A d-dimensional **hypercube** has 2^d vertices. When they are indexed by binary d-tuples $(a_d, a_{d-1}, \ldots, a_0)$, adjacent vertices are those whose tuples differ in one position. Thus, the 2D

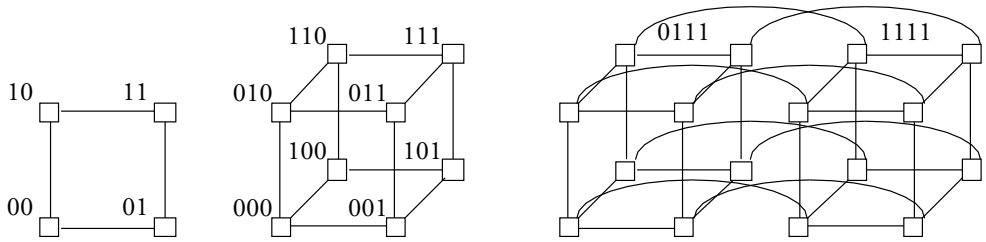

Figure 7.12 Hypercubes in two, three, and four dimensions.

hypercube is a square, the 3D hypercube is the traditional 3-cube, and the four-dimensional hypercube consists of two 3-cubes with edges between corresponding pairs of vertices. (See Fig. 7.12.) The d-dimensional hypercube is composed of two $(d-1)$-dimensional hypercubes in which each vertex in one hypercube has an edge to the corresponding vertex in the other. The degree of each vertex in a d-dimensional hypercube is d and its diameter is d as well.

While the hypercube is a very useful model for algorithm development, the construction of hypercube-based networks can be costly due to the high degree of the vertices. For example, each vertex in a hypercube with 4,096 vertices has degree 12; that is, each vertex is connected to 12 other vertices, and a total of 49,152 connections are necessary among the 4,096 processors. By contrast, a $2^6 \times 2^6$ 2D mesh has the same number of processors but at most 16,384 wires. The ratio between the number of wires in a d-dimensional hypercube and a square mesh with the same number of vertices is $d/4$. This makes it considerably more difficult to realize a hypercube of high dimensionality than a 2D mesh with a comparable number of vertices.

7.6.1 Embedding Arrays in Hypercubes

Given an algorithm designed for an array, we ask whether it can be efficiently realized on a hypercube network. The answer is positive. We show by induction that if d is even, a $2^{d/2} \times 2^{d/2}$ array can be embedded into a d-dimensional, 2^d-vertex hypercube and if d is odd, a $2^{(d+1)/2} \times 2^{(d-1)/2}$ array can be embedded into a d-dimensional hypercube. The base cases are $d = 2$ and $d = 3$.

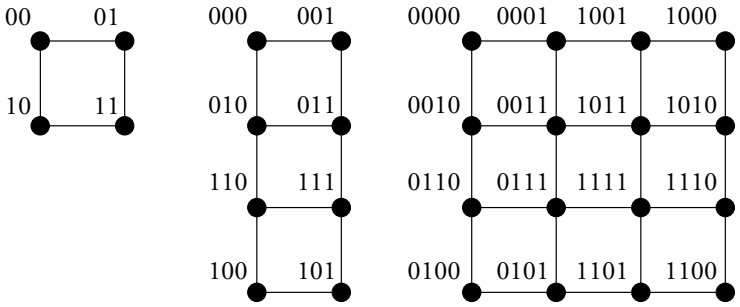

Figure 7.13 Mappings of 2×2, 4×2, and 4×4 arrays to two-, three-, and four-dimensional hypercubes. The binary tuples identify vertices of a hypercube.

When $d = 2$, a $2^{d/2} \times 2^{d/2}$ array is a 2×2 array that is itself a four-vertex hypercube. When $d = 3$, a $2^{(d+1)/2} \times 2^{(d-1)/2}$ array is a 4×2 array. (See Fig. 7.13, page 299.) It can be embedded into a three-dimensional hypercube by mapping the top and bottom 2×2 subarrays to the vertices of the two 2-cubes contained in the 3-cube. The edges between the two subarrays correspond directly to edges between vertices of the 2-cubes.

Applying the same kind of reasoning to the inductive hypothesis, we see that the hypothesis holds for all values of $d \geq 2$. If a 2D array is not of the form indicated, it can be embedded into such an array whose sides are a power of 2 by at most quadrupling the number of vertices.

7.6.2 Cube-Connected Cycles

A reasonable alternative to the hypercube is the **cube-connected cycles (CCC)** network shown in Fig. 7.14. Each of its vertices has degree 3, yet the graph has a diameter only a constant factor larger than that of the hypercube. The (d, r)-CCC is defined in terms of a d-dimensional hypercube when $r \geq d$. Let $(a_{d-1}, a_{d-2}, \ldots, a_0)$ and $(b_{d-1}, b_{d-2}, \ldots, b_0)$ be the indices of two adjacent vertices on the d-dimensional hypercube. Assume that these tuples differ in the jth component, $0 \leq j \leq d-1$; that is, $a_j = b_j \oplus 1$ and $a_i = b_i$ for $i \neq j$. Associated with vertex $(a_{d-1}, \ldots, a_p, \ldots, a_0)$ of the hypercube are the vertices $(p, a_{d-1}, \ldots, a_p, \ldots, a_0)$, $0 \leq p \leq r-1$, of the CCC that form a ring; that is, vertex $(p, a_{d-1}, \ldots, a_p, \ldots, a_0)$ is adjacent to vertices $((p+1) \bmod r, a_{d-1}, \ldots a_p, \ldots, a_0)$ and $((p-1) \bmod r, a_{d-1}, \ldots, a_p, \ldots, a_0)$. In addition, for $0 \leq p \leq d-1$, vertex $(p, a_{d-1}, \ldots, a_p, \ldots, a_0)$ is adjacent to vertex $(p, a_{d-1}, \ldots, a_p \oplus 1, \ldots, a_0)$ on the ring associated with vertex $(a_{d-1}, \ldots, a_p \oplus 1, \ldots, a_0)$ of the hypercube.

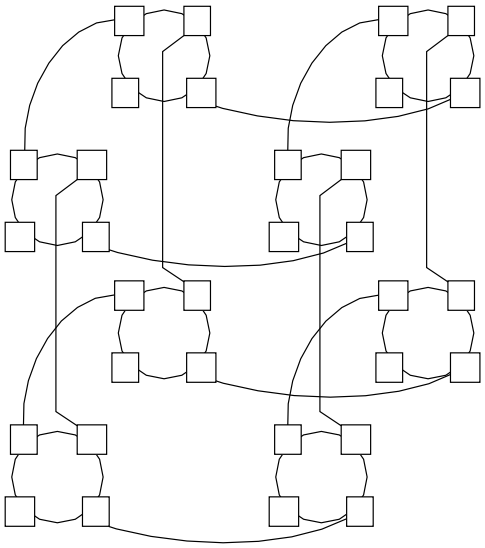

Figure 7.14 The cube-connected cycles network replaces each vertex of a d-dimensional hypercube with a ring of $r \geq d$ vertices in which each vertex is connected to its neighbor on the ring. The jth ring vertex, $0 \leq j \leq d-1$, is also connected to the jth ring vertex at an adjacent corner of the original hypercube.

The diameter of the CCC is at most $3r/2 + d$, as we now show. Given two vertices $\boldsymbol{v}_1 = (p, a_{d-1}, \ldots, a_0)$ and $\boldsymbol{v}_2 = (q, b_{d-1}, \ldots, b_0)$, let their hypercube addresses $\boldsymbol{a} = (a_{d-1}, \ldots, a_0)$ and $\boldsymbol{b} = (b_{d-1}, \ldots, b_0)$ differ in k positions. To move from $\boldsymbol{v}_1$ to $\boldsymbol{v}_2$, move along the ring containing $\boldsymbol{v}_1$ by decreasing processor numbers until reaching the next lower index at which $\boldsymbol{a}$ and $\boldsymbol{b}$ differ. (Wrap around to the highest index, if necessary.) Move from this ring to the ring whose hypercube address differs in this index. Move around this ring until arriving at the next lower indexed processor at which $\boldsymbol{a}$ and $\boldsymbol{b}$ differ. Continue in this fashion until reaching the ring with hypercube address $\boldsymbol{b}$. The number of edges traversed in this phase of the movement is at most one for each vertex on the ring plus at most one for each of the $k \leq d$ positions on which the addresses differ. Finally, move around the last ring toward the vertex $\boldsymbol{v}_2$ along the shorter path. This requires at most $r/2$ edge traversals. Thus, the maximal distance between two vertices, the diameter of the graph, is at most $3r/2 + d$.

7.7 Normal Algorithms

Normal algorithms on hypercubes are systolic algorithms with the property that in each cycle some bit position in an address is chosen and data is exchanged only between vertices whose addresses differ in this position. An operation is then performed on this data in one or both vertices. Thus, if the hypercube has three dimensions and the chosen dimension is the second, the following pairs of vertices exchange data and perform operations on them: $(0, 0, 0)$ and $(0, 1, 0)$, $(0, 0, 1)$ and $(0, 1, 1)$, $(1, 0, 0)$ and $(1, 1, 0)$, and $(1, 0, 1)$ and $(1, 1, 1)$. A **fully normal algorithm** is a normal algorithm that visits each of the dimensions of the hypercube in sequence. There are two kinds of fully normal algorithms, **ascending** and **descending algorithms**; ascending algorithms visit the dimensions of the hypercube in ascending order, whereas descending algorithms visit them in descending order. We show that many important algorithms are fully normal algorithms or combinations of ascending and descending algorithms. These algorithms can be efficiently translated into mesh-based algorithms, as we shall see.

The **fast Fourier transform** (FFT) (see Section 6.7.3) is an ascending algorithm. As suggested in the **butterfly graph** of Fig. 7.15, if each vertex at each level in the FFT graph on $n = 2^d$ inputs is indexed by a pair $(l, \boldsymbol{a})$, where $\boldsymbol{a}$ is a binary d-tuple and $0 \leq l \leq d$, then at level l pairs of vertices are combined whose indices differ in their lth component. (See Problem 7.14.) It follows that the FFT graph can be computed in levels on the d-dimensional hypercube by retaining the values corresponding to the column indexed by $\boldsymbol{a}$ in the hypercube vertex whose index is $\boldsymbol{a}$. It follows that the FFT graph has exactly the minimal connectivity required to execute an ascending fully normal algorithm. If the directions of all edges are reversed, the graph is exactly that needed for a descending fully normal algorithm. (The convolution function $f_{\text{conv}}^{(n,m)} : R^{n+m} \mapsto R^{n+m-1}$ over a commutative ring $\mathcal{R}$ can also be implemented as a normal algorithm in time $O(\log n)$ on an n-vertex hypercube, $n = 2^d$. See Problem 7.15.)

Similarly, because the graph of Batcher's bitonic merging algorithm (see Section 6.8.1) is the butterfly graph associated with the FFT, it too is a normal algorithm. Thus, two sorted lists of length $n = 2^d$ can be merged in $d = \log_2 n$ steps. As stated below, because the butterfly graph on 2^d inputs contains butterfly subgraphs on 2^k inputs, $k < d$, a recursive normal **sorting algorithm** can be constructed that sorts on the hypercube in $O(\log^2 n)$ steps. The reader is asked to prove the following theorem. (See Problem 6.29.)

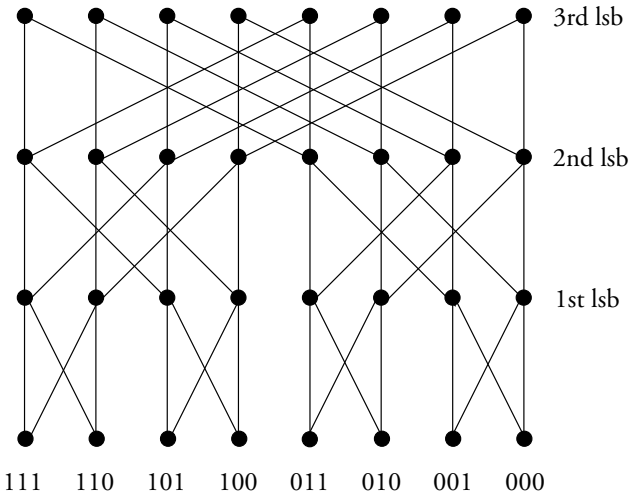

Figure 7.15 The FFT butterfly graph with column numberings. The predecessors of vertices at the kth level differ in their kth least significant bits.

THEOREM 7.7.1 *There exists a normal sorting algorithm on the p-vertex hypercube, $p = 2^d$, that sorts p items in time $O(\log^2 p)$.*

Normal algorithms can also be used to perform a **sum on the hypercube** and **broadcast on the hypercube**, as we show. We give an ascending algorithm for the first problem and a descending algorithm for the second.

7.7.1 Summing on the Hypercube

Let the hypercube be d-dimensional and let $\boldsymbol{a} = (a_{d-1}, a_{d-2}, \ldots, a_0)$ denote an address of a vertex. Associate with $\boldsymbol{a}$ the integer $|\boldsymbol{a}| = a_{d-1}2^{d-1} + a_{d-2}2^{d-2} + \cdots + a_0$. Thus, when $d = 3$, the addresses $\{0, 1, 2, \ldots, 7\}$ are associated with the eight 3-tuples $\{(0,0,0), (0,0,1), (0,1,0), \ldots, (1,1,1)\}$, respectively.

Let $V(|\boldsymbol{a}|)$ denote the value stored at the vertex with address $\boldsymbol{a}$. For each $(d-1)$ tuple $(a_{d-1}, \ldots, a_1)$, send to vertex $(a_{d-1}, \ldots, a_1, 0)$ the value stored at vertex $(a_{d-1}, \ldots, a_1, 1)$. In the summing problem we store at vertex $(a_{d-1}, \ldots, a_1, 0)$ the sum of the original values stored at vertices $(a_{d-1}, \ldots, a_1, 0)$ and $(a_{d-1}, \ldots, a_1, 1)$. Below we show the transmission (e.g. $V(0) \leftarrow V(1)$) and addition (e.g. $V(0) \leftarrow V(0) + V(1)$) that result for $d = 3$:

$$\begin{aligned} V(0) &\leftarrow V(1), & V(0) &\leftarrow V(0) + V(1) \\ V(2) &\leftarrow V(3), & V(2) &\leftarrow V(2) + V(3) \\ V(4) &\leftarrow V(5), & V(4) &\leftarrow V(4) + V(5) \\ V(6) &\leftarrow V(7), & V(6) &\leftarrow V(6) + V(7) \end{aligned}$$

For each $(d-2)$ tuple $(a_{d-1}, \ldots, a_2)$ we then send to vertex $(a_{d-1}, \ldots, a_2, 0, 0)$ the value stored at vertex $(a_{d-1}, \ldots, a_2, 1, 0)$. Again for $d = 3$, we have the following data transfers and additions:

$$V(0) \leftarrow V(2), \qquad V(0) \leftarrow V(0) + V(2),$$
$$V(4) \leftarrow V(6), \qquad V(4) \leftarrow V(4) + V(6),$$

We continue in this fashion until reaching the lowest dimension of the d-tuples at which point we have the following actions when $d = 3$:

$$V(0) \leftarrow V(4), \qquad V(0) \leftarrow V(0) + V(4)$$

At the end of this computation, $V(0)$ is the sum of the values stored in all vertices. This algorithm for computing $V(0)$ can be extended to any associative binary operator.

7.7.2 Broadcasting on the Hypercube

The broadcast operation is obtained by reversing the directions of each of the transmissions described above. Thus, in the example, $V(0)$ is sent to $V(4)$ in the first stage, in the second stage $V(0)$ and $V(4)$ are sent to $V(2)$ and $V(6)$, respectively, and in the last stage, $V(0)$, $V(2)$, $V(4)$, and $V(6)$ are sent to $V(1)$, $V(3)$, $V(5)$, and $V(7)$, respectively.

The algorithm given above to broadcast from one vertex to all others in a hypercube can be modified to broadcast to just the vertices in a subhypercube that is defined by those addresses $\boldsymbol{a} = (a_{d-1}, a_{d-2}, \ldots, a_0)$ in which all bits are fixed except for those in some k positions. For example, $\{(0,0,0), (0,1,0), (1,0,0), (1,1,0)\}$ are the vertices of a subhypercube of the three-dimensional hypercube (the rightmost bit is fixed). To broadcast to each of these vertices from $(0,1,0)$, say, on the first step send the message to its pair along the second dimension, namely, $(0,0,0)$. On the second step, let these pairs send messages to their pairs along the third dimension, namely, $(0,1,0) \rightarrow (1,1,0)$ and $(0,0,0) \rightarrow (1,0,0)$. This algorithm can be generalized to broadcast from any vertex in a hypercube to all other vertices in a subhypercube. Values at all vertices of a subhypercube can be associatively combined in a similar fashion.

The performance of these normal algorithms is summarized below.

THEOREM 7.7.2 *Broadcasting from one vertex in a d-dimensional hypercube to all other vertices can be done with a normal algorithm in $O(d)$ steps. Similarly, the associative combination of the values stored at the vertices of a d-dimensional hypercube can be done with a normal algorithm in $O(d)$ steps. Broadcasting and associative combining can also be done on the vertices of k-dimensional subcube of the d-dimensional hypercube in $O(k)$ steps with a normal algorithm.*

7.7.3 Shifting on the Hypercube

Cyclic shifting can also be done on a hypercube as a normal algorithm. For $n = 2^d$, consider shifting the n-tuple $\boldsymbol{x} = (x_{n-1}, \ldots, x_0)$ cyclically left by k places on a d-dimensional hypercube. If $k \leq n/2$ (see Fig. 7.16(a)), the largest element in the right half of $\boldsymbol{x}$, namely $x_{n/2-1}$, moves to the left half of $\boldsymbol{x}$. On the other hand, if $k > n/2$ (see Fig. 7.16(b)), $x_{n/2-1}$ moves to the right half of $\boldsymbol{x}$.

Thus, to shift $\boldsymbol{x}$ left cyclically by k places, $k \leq n/2$, divide $\boldsymbol{x}$ into two $(n/2)$-tuples, shift each of these tuples cyclically by k places, and then swap the rightmost k components of the two halves, as suggested in Fig. 7.16(a). The swap is done via edges across the highest

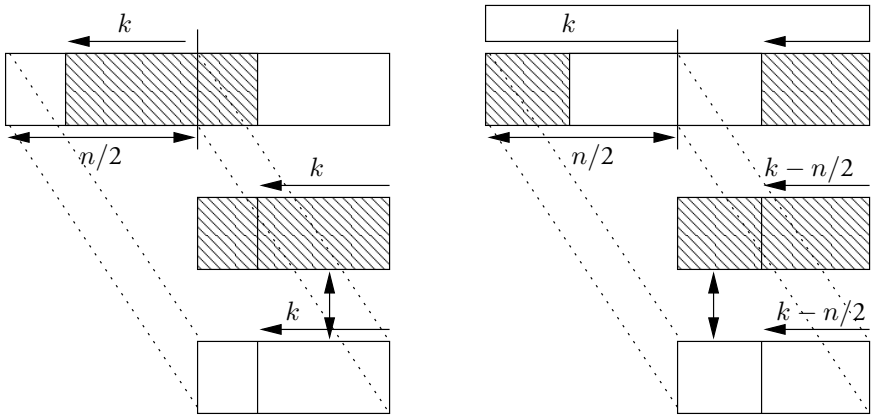

Figure 7.16 The two cases of a normal algorithm for cyclic shifting on a hypercube.

dimension of the hypercube. When $k > n/2$, cyclically shift each $(n/2)$-tuple by $k - n/2$ positions and then swap the high-order $n - k$ positions from each tuple across the highest dimension of the hypercube. We have the following result.

THEOREM 7.7.3 *Cyclic shifting of an n-tuple, $n = 2^d$, by any amount can be done recursively by a normal algorithm in $\log_2 n$ communication steps.*

7.7.4 Shuffle and Unshuffle Permutations on Linear Arrays

Because many important algorithms are normal and hypercubes are expensive to realize, it is preferable to realize normal algorithms on arrays. In this section we introduce the shuffle and unshuffle permutations, show that they can be used to realize normal algorithms, and then show that they can be realized on linear arrays. We use the unshuffle algorithms to map normal hypercube algorithms onto one- and two-dimensional meshes.

Let $\mathbb{N}(n) = \{0, 1, 2, \ldots, n-1\}$ and $n = 2^d$. The **shuffle permutation** $\pi^{(n)}_{\text{shuffle}} : \mathbb{N}(n) \mapsto \mathbb{N}(n)$ moves the item in position a to position $\pi^{(n)}_{\text{shuffle}}(a)$, where $\pi^{(n)}_{\text{shuffle}}(a)$ is the integer represented by the left cyclic shift of the d-bit binary number representing a. For example, when $n = 8$ the integer 3 is represented by the binary number 011 and its left cyclic shift is 110. Thus, $\pi^{(8)}_{\text{shuffle}}(3) = 6$. The shuffle permutation of the sequence $\{0, 1, 2, 3, 4, 5, 6, 7\}$ is the sequence $\{0, 4, 1, 5, 2, 6, 3, 7\}$. A shuffle operation is analogous to interleaving of the two halves of a sorted deck of cards. Figure 7.17 shows this mapping for $n = 8$.

The **unshuffle permutation** $\pi^{(n)}_{\text{unshuffle}} : \mathbb{N}(n) \mapsto \mathbb{N}(n)$ reverses the shuffle operation: it moves the item in position b to position a where $b = \pi^{(n)}_{\text{shuffle}}(a)$; that is, $a = \pi^{(n)}_{\text{unshuffle}}(b) = \pi_{\text{unshuffle}}(\pi_{\text{shuffle}}(a))$. Figure 7.18 shows this mapping for $n = 8$. The shuffle permutation is obtained by reversing the directions of edges in this graph.

An unshuffle operation can be performed on an n-cell linear array, $n = 2^d$, by assuming that the cells contain the integers $\{0, 1, 2, \ldots, n-1\}$ from left to right represented as d-bit binary integers and then sorting them by their least significant bit using a stable sorting algorithm. (A **stable sorting algorithm** is one that does not change the original order of keys

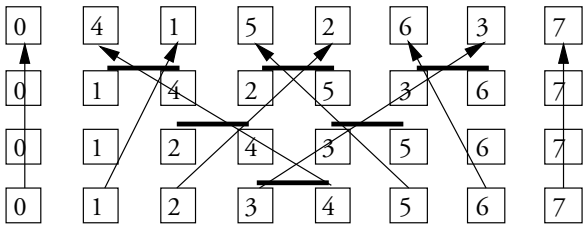

Figure 7.17 The shuffle permutation can be realized by a series of swaps of the contents of cells. The cells between which swaps are done have a heavy bar above them. The result of swapping cells of one row is shown in the next higher row, so that the top row contains the result of shuffling the bottom row.

with the same value.) When this is done, the sequence $\{0, 1, 2, 3, 4, 5, 6, 7\}$ is mapped to the sequence $\{0, 2, 4, 6, 1, 3, 5, 7\}$, the unshuffled sequence, as shown in Fig. 7.18. The integer b is mapped to the integer a whose binary representation is that of b shifted cyclically right by one position. For example, position 1 (001) is mapped to position 4 (100) and position 6 (110) is mapped to position 3 (011).

Since bubble sort is a stable sorting algorithm, we use it to realize the unshuffle permutation. (See Section 7.5.2.) In each phase keys (binary tuples) are compared based on their least significant bits. In the first phase values in positions i and $i + 1$ are compared for i even. The next comparison is between such pairs for i odd. Comparisons of this form continue, alternating between even and odd values for i, until the sequence is sorted. Since the first phase has no effect on the integers $\{0, 1, 2, \ldots, n - 1\}$, it is not done. Subsequent phases are shown in Fig. 7.18. Pairs that are compared are connected by a light line; a darker line joins pairs whose values are swapped. (See Problem 7.16.)

We now show how to implement efficiently a fully normal ascending algorithm on a linear array. (See Fig. 7.19.) Let the **exchange locations** of the linear array be locations i and $i + 1$ of the array for i even. Only elements in exchange locations are swapped. Swapping between the first dimension of the hypercube is done by swaps across exchange locations. To simulate exchanges across the second dimension, perform a shuffle operation (by reversing the order of the operations of Fig. 7.18) on each group of four elements. This places into exchange locations elements whose original indices differed by two. Performing a shuffle on eight, sixteen, etc.

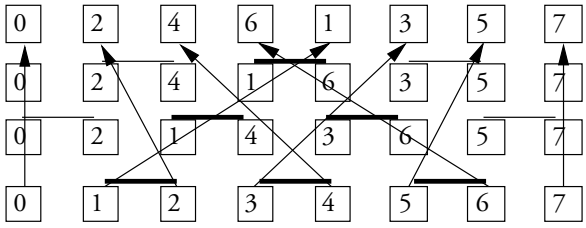

Figure 7.18 An unshuffle operation is obtained by bubble sorting the integers $\{0, 1, 2, \ldots, n - 1\}$ based on the value of their least significant bits. The cells with bars over them are compared. The first set of comparisons is done on elements in the bottom row. Those pairs with light bars contain integers whose values are in order.

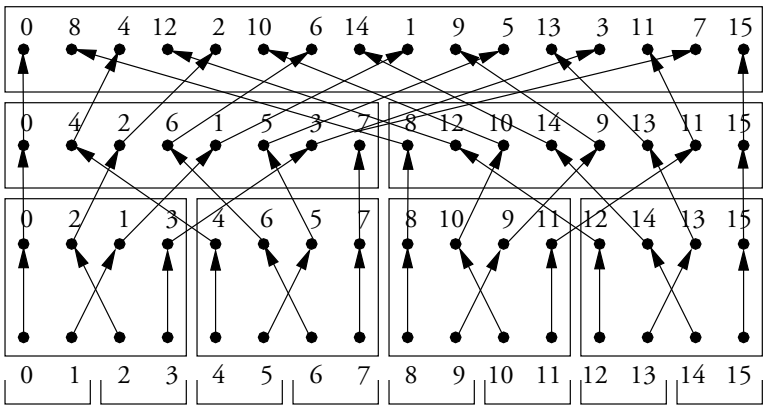

Figure 7.19 A normal ascending algorithm realized by shuffle operations on 2^k elements, $k = 2, 3, 4, \ldots$, places into exchange locations elements whose indices differ by increasing powers of two. Exchange locations are paired together.

positions places into exchange locations elements whose original indices differed by four, eight, etc. The proof of correctness of this result is left to the reader. (See Problem 7.17.)

Since a shuffle on $n = 2^d$ elements can be done in $2^{d-1} - 1$ steps on a linear array with n cells (see Theorem 7.5.2), it follows that this fully normal ascending algorithm uses $T(n) = \phi(d)$ steps, where $T(2) = \phi(1) = 0$ and

$$\phi(d) = \phi(d-1) + 2^{d-1} - 1 = 2^d - d - 1$$

Do a fully normal descending algorithm by a shuffle followed by its steps in reverse order.

THEOREM 7.7.4 *A fully normal ascending (descending) algorithm that runs in $d = \log_2 n$ steps on a d-dimensional hypercube containing 2^d vertices can be realized on a linear array of $n = 2^d$ elements with $T(n) = n - \log_2 n - 1$ (2T(n)) additional parallel steps.*

From the discussion of Section 7.7 it follows that broadcasting, associative combining, and the FFT algorithm can be executed on a linear array in $O(n)$ steps because each can be implemented as a normal algorithm on the n-vertex hypercube. Also, a list of n items can be sorted on a linear array in $O(n)$ steps by translating Batcher's sorting algorithm based on bitonic merging, a normal sorting algorithm, to the linear array. (See Problem 7.20.)

7.7.5 Fully Normal Algorithms on Two-Dimensional Arrays

We now consider the execution of a normal algorithm on a rectangular array. We assume that the $n = 2^{2d}$ vertices of a $2d$-dimensional hypercube are mapped onto an $m \times m$ mesh, $m = 2^d$, in row-major order. Since each cell is indexed by a pair consisting of row and column indices, (r, c), and each of these satisfies $0 \le r \le m-1$ and $0 \le c \le m-1$, they can each be represented by a d-bit binary number. Let $\boldsymbol{r}$ and $\boldsymbol{c}$ be these binary numbers. Thus cell (r, c) is indexed by the $2d$-bit binary number $\boldsymbol{rc}$.

Cells in positions (r, c) and $(r, c+1)$ have associated binary numbers that agree in their d most significant positions. Cells in positions (r, c) and $(r+1, c)$ have associated binary

numbers that agree in their d least significant positions. To simulate a normal hypercube algorithm on the 2D mesh, in each row simulate a normal hypercube algorithm on 2^d vertices after which in each column simulate a normal hypercube algorithm on 2^d vertices. The correctness of this procedure follows because every adjacent pair of vertices of the simulated hypercube is at some time located in adjacent cells of the 2D array.

From Theorem 7.7.4 it follows that hypercube exchanges across the lower d dimensions can be simulated in time proportional to the length of a row, that is, in time $O(\sqrt{n})$. Similarly, it also follows that hypercube exchanges across the higher d dimensions can be simulated in time proportional to $O(\sqrt{n})$. We summarize this result below.

THEOREM 7.7.5 *A fully normal $2d$-dimensional hypercube algorithm (ascending or descending), $n = 2^{2d}$, can be realized in $O(\sqrt{n})$ steps on an $\sqrt{n} \times \sqrt{n}$ array of cells.*

It follows from the discussion of Section 7.7 that broadcasting, associative combining, and the FFT algorithm can be executed on a 2D mesh in $O(\sqrt{n})$ steps because each can be implemented as a normal algorithm on the n-vertex hypercube.

Also, a list of n items can be sorted on an $\sqrt{n} \times \sqrt{n}$ array in $O(\sqrt{n})$ steps by translating a normal merging algorithm to the $\sqrt{n} \times \sqrt{n}$ array and using it recursively to create a sorting network. (See Problem 7.21.) No sorting algorithm can sort in fewer than $2\sqrt{m} - 2$ steps on an $\sqrt{m} \times \sqrt{m}$ array because whatever element is positioned in the lower right-hand corner of the array could originate in the upper left-hand corner and have to traverse at least $2\sqrt{m} - 2$ edges to arrive there.

7.7.6 Normal Algorithms on Cube-Connected Cycles

Consider now processors connected as a d-dimensional cube-connected cycle (CCC) network in which each ring has $r = 2^k \geq d$ processors. In particular, let r be the smallest power of 2 greater than or equal to d, so that $d \leq r < 2d$. (Thus $k = O(\log d)$.) We call such a CCC network a **canonical CCC network** on n vertices. It has $n = r2^d$ vertices, $d2^d \leq n < (2d)2^d$. (Thus $d = O(\log n)$.) We show that a fully normal algorithm can be executed efficiently on such CCC networks.

Let each ring of the CCC network be indexed by a d-tuple corresponding to the corner of the hypercube at which it resides. Let each processor be indexed by a $(d + k)$-tuple in which the d low-order bits are the ring index and the k high-order bits specify the position of a processor on the ring.

A fully normal algorithm on a canonical CCC network is implemented in two phases. In the first phase, the ring is treated as an array and a fully normal algorithm on the k high-order bits is simulated in $O(d)$ steps. In the second phase, exchanges are made across hypercube edges. Rotate the elements on each ring so that ring processors whose k-bit indices are **0** (call these the **lead elements**) are adjacent along the first dimension of the original hypercube. Exchange information between them. Now rotate the rings by one position so that lead elements are adjacent along the second dimension of the original hypercube. The elements immediately behind the lead elements on the rings are now adjacent along the first hypercube dimension and are exchanged in parallel with the lead elements. (This simultaneous execution is called **pipelining**.) Subsequent rotations of the rings place successive ring elements in alignment along increasing bit positions. After $O(d)$ rotations all exchanges are complete. Thus, a total of $O(d)$ time steps suffice to execute a fully normal algorithm. We have the following result.

THEOREM 7.7.6 *A fully normal algorithm (ascending or descending) for an n-vertex hypercube can be realized in $O(\log n)$ steps on a canonical n-vertex cube-connected cycle network.*

Thus, a fully normal algorithm on an n-vertex hypercube can be simulated on a CCC network in time proportional to the time on the hypercube. However, the vertices of the CCC have bounded degree, which makes them much easier to realize in hardware than high-degree networks.

7.7.7 Fast Matrix Multiplication on the Hypercube

Matrix multiplication can be done more quickly on the hypercube than on a two-dimensional array. Instead of $O(n)$ steps, only $O(\log n)$ steps are needed, as we show.

Consider the multiplication of $n \times n$ matrices A and B for $n = 2^r$ to produce the product matrix $C = A \times B$. We describe a normal systolic algorithm to multiply these matrices on a d-dimensional hypercube, $d = 3r$.

Since $d = 3r$, the vertices of the d-dimensional hypercube are addressed by a binary $3r$-tuple, $\boldsymbol{a} = (a_{3r-1}, a_{3r-2}, \ldots, a_0)$. Let the r least significant bits of $\boldsymbol{a}$ denote an integer i, let the next r lsb's denote an integer j, and let the r most significant bits denote an integer k. Then, we have $|\boldsymbol{a}| = kn^2 + jn + i$ since $n = 2^r$. Because of this identity, we represent the address $\boldsymbol{a}$ by the triple (i, j, k). We speak of the processor $P_{i,j,k}$ located at the vertex (i, j, k) of the d-dimensional hypercube, $d = 3r$. We denote by $HC_{i,j,-}$ the subhypercube in which i and j are fixed and by $HC_{i,-,k}$ and $HC_{-,j,k}$ the subhypercubes in which the two other pairs of indices are fixed. There are 2^{2r} subhypercubes of each kind.

We assume that each processor $P_{i,j,k}$ contains three local variables, $A_{i,j,k}$, $B_{i,j,k}$, and $C_{i,j,k}$. We also assume that initially $A_{i,j,0} = a_{i,j}$ and $B_{i,j,0} = b_{i,j}$, where $0 \le i, j \le n-1$. The multiplication algorithm has the following five phases:

a) For each subhypercube $HC_{i,j,-}$ and for $1 \le k \le n-1$, broadcast $A_{i,j,0}$ (containing $a_{i,j}$) to $A_{i,j,k}$ and $B_{i,j,0}$ (containing $b_{i,j}$) to $B_{i,j,k}$.

b) For each subhypercube $HC_{i,-,k}$ and for $0 \le j \le n-1$, $j \ne k$, broadcast $A_{i,k,k}$ (containing $a_{i,k}$) to $A_{i,j,k}$.

c) For each subhypercube $HC_{-,j,k}$ and for $0 \le i \le n-1$, $i \ne k$, broadcast $B_{k,j,k}$ (containing $b_{k,j}$) to $B_{i,j,k}$.

d) At each processor $P_{i,j,k}$ compute $C_{i,j,k} = A_{i,j,k} \cdot B_{i,j,k} = a_{i,k} b_{k,j}$.

e) At processor $P_{i,j,0}$ compute the sum $C_{i,j,0} = \sum_k C_{i,j,k}$ ($C_{i,j,0}$ now contains $c_{i,j} = \sum_k a_{i,k} b_{k,j}$).

From Theorem 7.7.2 it follows that each of these five steps can be done in $O(r)$ steps, where $r = \log_2 n$. We summarize this result below.

THEOREM 7.7.7 *Two $n \times n$ matrices, $n = 2^r$, can be multiplied by a normal systolic algorithm on a d-dimensional hypercube, $d = 3r$, with n^3 processors in $O(\log n)$ steps. All normal algorithms for $n \times n$ matrix multiplication require $\Omega(\log n)$ steps.*

Proof The upper bound follows from the construction. The lower bound follows from the observation that each processor that is participating in the execution of a normal algorithm

combines two values, one that it owns and one owned by one of its neighbors. Thus, if t steps are executed to compute a value, that value cannot depend on more than 2^t other values. Since each entry in an $n \times n$ product matrix is a function of $2n$ other values, t must be at least $\log_2(2n)$. ■

The lower bound stated above applies only to normal algorithms. If a non-normal algorithm is used, each processor can combine up to d values. Thus, after k steps, up to d^k values can be combined. If $2n$ values must be combined, as in $n \times n$ matrix multiplication, then $k \geq \log_d(2n) = (\log_2 2n)/\log_2 d$. If an n^3-processor hypercube is used for this problem, $d = 3\log_2 n$ and $k = \Omega(\log n/\log\log n)$.

The normal matrix multiplication algorithm described above can be translated to linear arrays and 2D meshes using the mappings based on the shuffle and unshuffle operations. The 2D mesh version has a running time $O(\sqrt{n}\log n)$, which is inferior to the running time of the algorithm given in Section 7.5.3.

7.8 Routing in Networks

A topic of major concern in the design of distributed memory machines is **routing**, the task of transmitting messages among processors via nodes of a network. Routing becomes challenging when many messages must travel simultaneously through a network because they can produce congestion at nodes and cause delays in the receipt of messages.

Some routing networks are designed primarily for the **permutation-routing problem**, the problem of establishing a one-to-one correspondence between n senders and n receivers. (A processor can be both a sender and receiver.) Each sender sends one message to a unique receiver and each receiver receives one message from a unique sender. (We examine in Section 7.9.3 routing methods when the numbers of senders and receivers differ and more than one message can be received by one processor.) If many messages are targeted at one receiver, a long delay will be experienced at this receiver. It should be noted that network congestion can occur at a node even when messages are uniformly distributed throughout the network, because many messages may have to pass through this node to reach their destinations.

7.8.1 Local Routing Networks

In a **local routing network** each message is accompanied by its destination address. At each network node (**switch**) the routing algorithm, using only these addresses and not knowing the global state of the network, finds a path for messages.

A sorting network, suitably modified to transmit messages, is a local permutation-routing network. Batcher's bitonic sorting network described in Section 6.8.1 will serve as such a network. As mentioned in Section 7.7, this network can be realized as a normal algorithm on a hypercube, with running time on an n-vertex hypercube $O(\log^2 n)$. (See Problem 6.28.) On the two-dimensional mesh its running time is $O(\sqrt{n})$ (see Problem 7.21), whereas on the linear array it is $O(n)$ (see Problem 7.20).

Batcher's bitonic sorting network is **data-oblivious**; that is, it performs the same set of operations for all values of the input data. The outcomes of these operations are data-dependent, but the operations themselves are data-independent. Non-oblivious sorting algorithms perform operations that depend on the values of the input data. An example of a local non-

oblivious algorithm is one that sends a message from the current network node to the neighboring node that is closest to the destination.

7.8.2 Global Routing Networks

In a **global routing network**, knowledge of the destinations of all messages is used to set the network switches and select paths for the messages to follow. A global permutation-routing network realizes permutations of the destination addresses. We now give an example of such a network, the Beneš permutation network.

A permutation network is constructed of two-input, two-output switches. Such a switch either passes its inputs, labeled A and B, to its outputs, labeled X and Y, or it swaps them. That is, the switch is set so that either X = A and Y = B or X = B and Y = A. A **permutation network** on n inputs and n outputs is a directed acyclic graph of these switches such that for each permutation of the n inputs, switches can be set to create n disjoint paths from the n inputs to the n outputs.

A **Beneš permutation network** is shown in Fig. 7.20. This graph is produced by connecting two copies of an FFT graph on 2^{k-1} inputs back to back and replacing the nodes by switches and edges by pairs of edges. (FFT graphs are described in Section 6.7.3.) It follows that a Beneš permutation network on n inputs can be realized by a normal algorithm

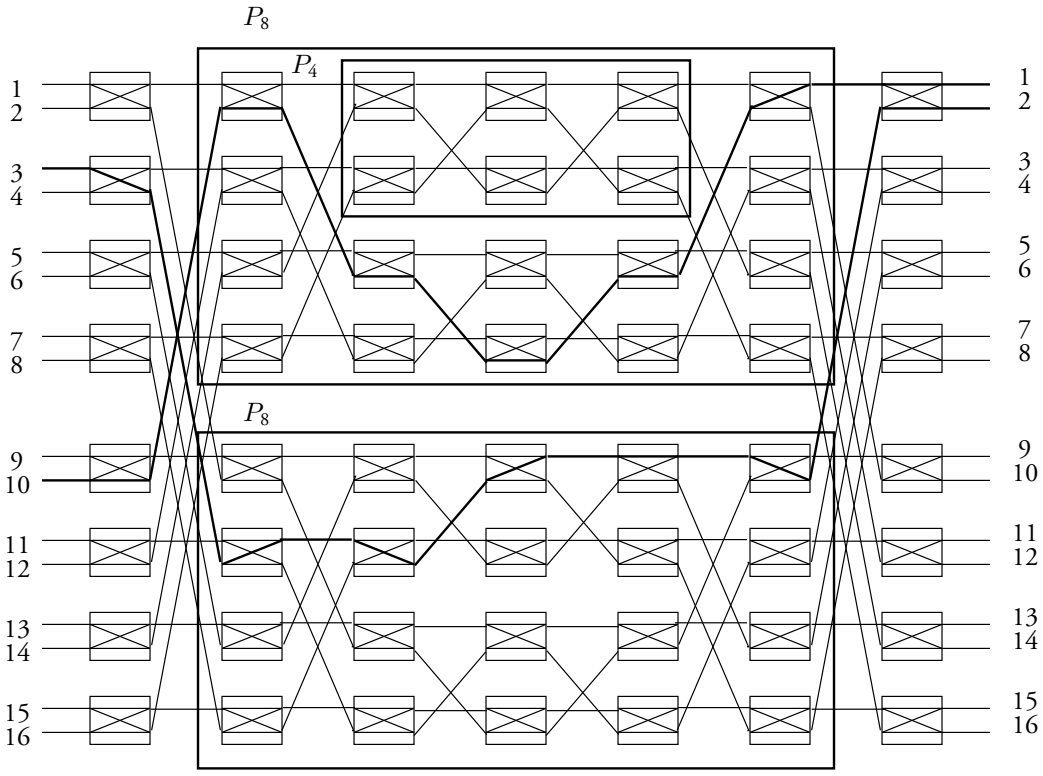

Figure 7.20 A Beneš permutation network.

that executes $O(\log n)$ steps. Thus, a permutation is computed much more quickly (in time $O(\log n)$) with the Beneš offline permutation network than it can be done on Batcher's online bitonic sorting network (in time $O(\log^2 n)$). However, the Beneš network requires time to collect the destinations at some central location, compute the switch settings, and transmit them to the switches themselves.

To understand how the Beneš network works, we provide an alternative characterization of it. Let P_n be the Beneš network on n inputs, $n = 2^k$, defined as back-to-back FFT graphs with nodes replaced by switches. Then P_n may be defined recursively, as suggested in Fig. 7.20. P_n is obtained by making two copies of $P_{n/2}$, placing $n/2$ copies of a two-input, two-output switch at the input and the same number at the output. For $1 \leq i \leq n/4$ ($n/4+1 \leq i \leq n/2$) the top output of switch i is connected to the top input of the ith switch in the upper (lower) copy of $P_{n/2}$ and the bottom output is connected to the bottom input of the ith switch in the lower (upper) copy of $P_{n/2}$. The connections of output switches are the mirror image of the connections of the input switches.

Consider the Beneš network P_2. It consists of a single switch and generates the two possible permutations of the inputs. We show by induction that P_n generates all $n!$ permutations of its n inputs. Assume that this property holds for $n = 2, 4, \ldots, 2^{k-1}$. We show that it holds for $m = 2^k$. Let $\boldsymbol{\pi} = (\pi(1), \pi(2), \ldots, \pi(m))$ be an arbitrary permutation to be realized by P_m. This means that the ith input must be connected to the $\pi(i)$th output. Suppose that $\pi(3)$ is 2, as shown in Fig. 7.20. We can arbitrarily choose to have the third input pass through the first or second copy of $P_{m/2}$. We choose the second. The path taken through the second copy of $P_{m/2}$ must emerge on its second output so that it can then pass to the first switch in the column of output switches. This output switch must pass its inputs without swapping them. The other output of this switch, namely 1, must arrive via a path through the first copy of $P_{m/2}$ and emerge on its first output. To determine the input at which it must arrive, we find the input of P_m associated with the output of 1 and set its switch so that it is directed to the first copy of $P_{m/2}$. Since the other input to this input switch must go to the other copy of $P_{m/2}$, we follow its path through P_m to the output and then reason in the same way about the other output at the output switch at which it arrives. If by tracing paths back and forth this way we do not exhaust all inputs and outputs, we pick another input and repeat the process until all inputs have been routed to outputs.

Now let's determine the number of switches, $S(k)$, in a Beneš network P_n on $n = 2^k$ inputs. It follows that $S(1) = 1$ and

$$S(k) = 2S(k-1) + 2^k$$

It is straightforward to show that $S(k) = (k - \frac{1}{2})2^k = n(\log_2 n - \frac{1}{2})$.

Although a global permutation network sends messages to their destinations more quickly than a local permutation network, the switch settings must be computed and distributed globally, both of which impose important limitations on the time to realize particular permutations.

7.9 The PRAM Model

The **parallel random-access machine (PRAM)** (see Fig. 7.21), the canonical structured parallel machine, consists of a bounded set of processors and a common memory containing a potentially unlimited number of words. Each processor is similar to the random-access machine (RAM) described in Section 3.4 except that its CPU can access locations in both its local

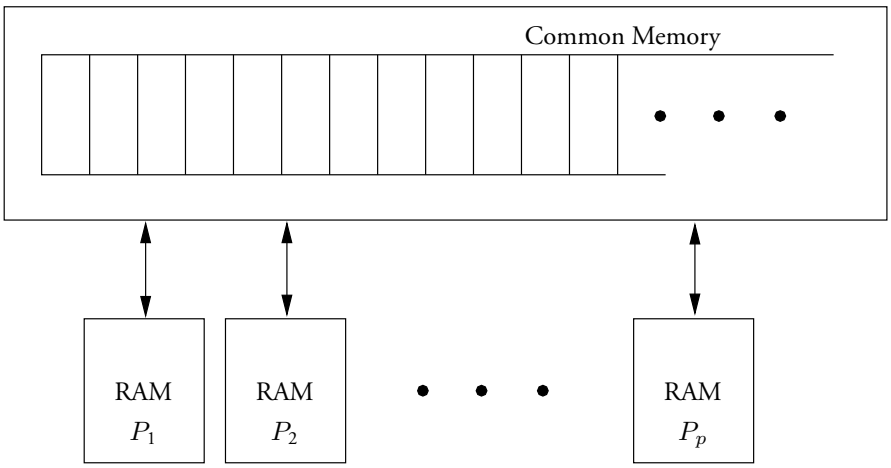

Figure 7.21 The PRAM consists of synchronous RAMs accessing a common memory.

random-access memory and the common memory. During each PRAM step, the RAMs execute the following steps in synchrony: they (a) read from the common memory, (b) perform a local computation, and (c) write to the common memory. Each RAM has its own program and program counter as well as a unique identifying number id_j that it can access to make processor-dependent decisions. The PRAM is primarily an abstract programming model, not a machine designed to be built (unlike mesh-based computers, for example).

The power of the PRAM has been explored by considering a variety of assumptions about the length of local computations and the type of instruction allowed. In designing parallel algorithms it is generally assumed that each local computation consists of a small number of instructions. However, when this restriction is dropped and the PRAM is allowed an unlimited number of computations between successive accesses to the common memory (the **ideal PRAM**), the information transmitted between processors reflects the minimal amount of information that must be exchanged to solve a problem on a parallel computer.

Because the size of memory words is potentially unbounded, very large numbers can be generated very quickly on a PRAM if a RAM can multiply and divide integers and perform vector operations. This allows each RAM to emulate a parallel machine with an unbounded number of processors. Since the goal is to understand the power of parallelism, however, this form of hidden parallelism is usually disallowed, either by not permitting these instructions or by assuming that in t steps a PRAM generates numbers whose size is bounded by a polynomial in t. To simplify the discussion, we limit instructions in a RAM's repertoire to addition, subtraction, vector comparison operations, conditional branching, and shifts by fixed amounts. We also allow load and store instructions for moving words between registers, local memories, and the common memory. These instructions are sufficiently rich to compute all computable functions.

As yet we have not specified the conditions under which access to the common memory occurs in the first and third substeps of each PRAM step. If access by more than one RAM to the same location is disallowed, access is **exclusive**. If this restriction does not apply, access is **concurrent**. Four combinations of these classifications apply to reading and writing. The strongest

restriction is placed on the **Exclusive Read/Exclusive Write (EREW) PRAM**, with successively weaker restrictions placed on the **Concurrent Read/Exclusive Write (CREW) PRAM**, the **Exclusive Read/Concurrent Write (ERCW) PRAM**, and the **Concurrent Read/Concurrent Write (CRCW) PRAM**. When concurrent writing is allowed, conflicts are resolved in one of the following ways: a) the **COMMON model** requires that all RAMs writing to a common location write the same value, b) the **ARBITRARY model** allows an arbitrary value to be written, and c) the **PRIORITY model** writes into the common location the value being written by the lowest numbered RAM.

Observe that any algorithm written for the COMMON CRCW PRAM runs without change on the ARBITRARY CRCW PRAM. Similarly, an ARBITRARY CRCW PRAM algorithm runs without change on the PRIORITY CRCW PRAM. Thus, the latter is the most powerful of the PRAM models.

In performing a computation on a PRAM it is typically assumed that the input is written in the lowest numbered locations of the common memory. PRAM computations are characterized by p, the **number of processors** (RAMs) in use, and T (**time**), the number of PRAM steps taken. Both measures are usually stated as a function of the size of a problem instance, namely m, the number of input words, and n, their total length in bits.

After showing that tree, array, and hypercube algorithms translate directly to a PRAM algorithm with no loss in efficiency, we explore the power of concurrency. This is followed by a brief discussion of the simulation of a PRAM on a hypercube and a circuit on a CREW PRAM. We close by referring the reader to connections established between PRAMs and circuits and to the discussion of serial space and parallel time in Chapter 8.

7.9.1 Simulating Trees, Arrays, and Hypercubes on the PRAM

We have shown that 1D arrays can be embedded into 2D meshes and that d-dimensional meshes can be embedded into hypercubes while preserving the neighborhood structure of the first graph in the second. Also, we have demonstrated that any balanced tree algorithm can be simulated as a normal algorithm on a hypercube. As a consequence, in each case, an algorithm designed for the first network carries over to the second without any increase in the number of steps executed. We now show that normal hypercube algorithms are efficiently simulated on an EREW PRAM.

With each d-dimensional hypercube processor, associate an EREW PRAM processor and a reserved location in the common memory. In a normal algorithm each hypercube processor communicates with its neighbor along a specified direction. To simulate this communication, each associated PRAM processor writes the data to be communicated into its reserved location. The processor for which the message is destined knows which hypercube neighbor is providing the data and reads the value stored in its associated memory location.

When a hypercube algorithm is not normal, as many as $d-1$ neighbors can send messages to one processor. Since EREW PRAM processors can access only one cell per unit time, simulation of the hypercube can require a running time that is about d times that of the hypercube.

THEOREM 7.9.1 *Every T-step normal algorithm on the d-dimensional, n-vertex hypercube, $n = 2^d$, can be simulated in $O(T)$ steps on an n-processor EREW PRAM. Every T-step hypercube algorithm, normal or not, can be simulated in $O(Td)$ steps.*

An immediate consequence of Theorems 7.7.1 and 7.9.1 is that a list of n items can be sorted on an n-processor PRAM in $O(\log^2 n)$ steps by a normal oblivious algorithm. Data-dependent sorting algorithms for the hypercube exist with running time $O(\log n)$.

It also follows from Section 7.6.1 that algorithms for trees, linear arrays, and meshes translate directly into PRAM algorithms with the same running time as on these less general models. Of course, the superior connectivity between PRAM processors might be used to produce faster algorithms.

7.9.2 The Power of Concurrency

The CRCW PRAM is a very powerful model. As we show, any Boolean function can be computed with it in a constant number of steps if a sufficient number of processors is available. For this reason, the CRCW PRAM is of limited interest: it represents an extreme that does not reflect reality as we know it. The CREW and EREW PRAMs are more realistic. We first explore the power of the CRCW and then show that an EREW PRAM can simulate a p-processor CRCW PRAM with a slowdown by a factor of $O(\log^2 p)$.

THEOREM 7.9.2 *The CRCW PRAM can compute an arbitrary Boolean function in four steps.*

Proof Given a Boolean function $f : \mathcal{B}^n \mapsto \mathcal{B}$, represent it by its disjunctive normal form; that is, represent it as the OR of its minterms where a minterm is the AND of each literal of f. (A literal is a variable, x_i, or its complement, $\overline{x}_i$.) Assume that each variable is stored in a separate location in the common memory.

Given a minterm, we show that it can be computed by a CRCW PRAM in two steps. Assign one location in the common memory to the minterm and initialize it to the value 1. Assign one processor to each literal in the minterm. The processor assigned to the jth literal reads the value of the jth variable from the common memory. If the value of the literal is 0, this processor writes the value 0 to the memory location associated with the minterm. Thus, the minterm has value 1 exactly when each literal has value 1. Note that these processors read concurrently with processors associated with other minterms and may write concurrently if more than one of their literals has value 0.

Now assume that a common memory location has been reserved for the function itself and initialized to 0. One processor is assigned to each minterm and if the value of its minterm is 1, it writes the value 1 in the location associated with the function. Thus, in two more steps the function f is computed. ■

Given the power of concurrency, especially as applied to writing, we now explore the cost in performance of not allowing concurrency, whether in reading or writing.

THEOREM 7.9.3 *A p-processor priority CRCW PRAM can be simulated by a p-processor EREW PRAM with a slowdown by a factor equal to the time to sort p elements on this machine. Consequently, this simulation can be done by a normal algorithm with a slowdown factor of $O(\log^2 p)$.*

Proof The jth EREW PRAM processor simulates a memory access by the jth CRCW PRAM processor by first writing into a special location, M_j, a pair $(\boldsymbol{a}_j, j)$ indicating that processor j wishes to access (read or write) location $\boldsymbol{a}_j$. If processors are writing to common memory, the value to be written is attached to this pair. If processors are reading from common memory, a return message containing the requested value is provided. If a processor chooses not to access any location, a dummy address larger than all other addresses is used for

$\boldsymbol{a}_j$. The contents of the locations $M_1, M_2, \ldots, M_p$ are sorted, which creates a subsequence in which pairs with a common address occur together and within which the pairs are sorted by processor numbers. From Theorem 7.7.1 it follows that this step can be performed in time $O(\log^2 p)$ by a normal algorithm. So far no concurrent reads or writes occur.

A processor is now assigned to each pair in the sorted sequence. We consider two cases: a) processors are reading from or b) writing to common memory. Each processor now compares the address of its pair to that of the preceding pair. If a processor finds these addresses to be different and case a holds, it reads the item in common memory and sets a flag bit to 1; all other processors except the first set their flag bits to 0; the first sets its bit to 1. (This bit is used later to distribute the value that was read.) However, if case b holds instead, the processor writes its value. Since this processor has the lowest index of all processors and the priority CRCW is the strongest model, the value written is the same value written by either the common or arbitrary CRCW models.

Returning now to case a, the flag bits mark the first pair in each subsequence of pairs that have the same address in the common memory. Associated with the leading pair is the value read at this address. We now perform a segmented prefix computation using as the associative rule the copy-right operation. (See Problem 2.20.) It distributes to each pair $(\boldsymbol{a}_j, j)$ the value the processor wished to read from the common memory. By Problem 2.21 this problem can be solved by a p-processor EREW PRAM in $O(\log p)$ steps. The pairs and their accompanying value are then sorted by the processor number so that the value read from the common memory is in a location reserved for the processor that requested the value. ■

7.9.3 Simulating the PRAM on a Hypercube Network

As stated above, each PRAM cycle involves reading from the global memory, performing a local computation, and writing to the common memory. Of course, a processor need not access common memory when given the chance. Thus, to simulate a PRAM on a network computer, one has to take into account the fact that not all PRAM processors necessarily read from or write to common memory locations on each cycle.

It is important to remember that the latency of network computers can be large. Thus, for the simulation described below to be useful, each PRAM processor must be able to do a lot of work between network accesses.

The EREW PRAM is simulated on a network computer by executing three phases, two of which correspond to reading and writing common memory. (To simulate the CRCW PRAM, we need only add the time given above to simulate a CRCW PRAM by an EREW PRAM.) We simulate an access to common memory by routing a message over the network to the site containing the simulated common memory location. It follows that a message must contain the name of a site as well as the address of a memory location at that site. If the simulated access is a memory read, a return message is generated containing the value of the memory location. If it is a memory write, the transmitted message must also contain the datum to write into the memory location. We assume that the sites are numbered consecutively from 1 to p, the number of processors.

The first problem to be solved is the routing of messages from source to destination processors. This routing problem was partially addressed in Section 7.8. The new wrinkle here is that the mapping from source to destination sites defined by a set of messages is not necessarily a permutation. Not all sources may send a message and not all destinations are guaranteed to

receive only one message. In fact, some destination may be sent many messages, which can result in their waiting a long time for receipt.

To develop an appreciation for the various approaches to this problem, we describe an algorithm that distributes messages from sources to destinations, though not as efficiently as possible. Each processor prepares a message to be sent to other processors. Processors not accessing the common memory send messages containing dummy site addresses larger than any other address. All messages are sorted by destination address cooperatively by the processors. As seen in Theorem 7.7.1, they can be sorted by a normal algorithm on an p-vertex hypercube, $p = 2^d$, in $O(\log^2 p)$ steps using Batcher's bitonic sorting network described in Section 6.8.1. The $k \leq p$ non-dummy messages are the first k messages in this sorted list. If the sites at which these messages reside after sorting are the sites for which they were destined, the message routing problem is solved. Unfortunately, this is generally not the case.

To route the messages from their positions in the sorted list to their destinations, we first identify duplicates of destination addresses and compute D, the maximum number of duplicates. We then route messages in D stages. In each stage at most one of the D duplicates of each message is routed to its destination. To identify duplicates, we assign a processor to each message in the sorted list that compares its destination site with that of its predecessor, setting a flag bit to 0 if equal and to 1 otherwise. To compare destinations, move messages to adjacent vertices on the hypercube, compare, and then reverse the process. (Move them by sorting by appropriate addresses.) The first processor also sets its flag bit to 1. A segmented integer addition prefix operation that segments its messages with these flag bits assigns to each message an integer (a **priority**) between 1 and D that is q if the site address of this message is the qth such address. (Prefix computations can be done on a p-vertex hypercube in $O(\log p)$ steps. See Problem 7.23.) A message with priority q is routed to its destination in the qth stage. An unsegmented prefix operation with max as the operator is then used to determine D.

In the qth stage, $1 \leq q \leq D$, all non-dummy messages with priority q are routed to their destination site on the hypercube as follows:

a) one processor is assigned to each message;

b) each such processor computes the **gap**, the difference between the destination and current site of its message;

c) each gap $\boldsymbol{g}$ is represented as a binary d-tuple $\boldsymbol{g} = (g_{d-1}, \ldots, g_0)$;

d) For $t = d-1, d-2, \ldots, 0$, those messages whose gap contains 2^t are sent to the site reached by crossing the tth dimension of the hypercube.

We show that in at most $O(D \log p)$ steps all messages are routed to their destinations. Let the sorted message sites form an ascending sequence. If there are k non-dummy messages, let gap_i, $0 \leq i \leq k-1$, be the gap of the ith message. Observe that these gaps must also form a nondecreasing sequence. For example, shown below is a sorted set of destinations and a corresponding sequence of gaps:

gap_i	1	1	2	2	3	6	7	8							
dest_i	1	2	4	5	7	11	13	15							
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	15

All the messages whose gaps contain 2^{d-1} must be the last messages in the sequence because the gaps would otherwise be out of order. Thus, advancing messages with these gaps by

2^{d-1} positions, which is done by moving them across the largest dimension of the hypercube, advances them to positions in the sequence that cannot be occupied by any other messages, even after these messages have been advanced by their full gaps. For example, shown below are the positions of the messages given above after those whose gaps contain 8 and 4 have been moved by this many positions:

dest_i	1	2	4	5	7					11	13					15
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	15	

Repeating this argument on subsequent smaller powers of 2, we find that no two messages that are routed in a given stage occupy the same site. As a consequence, after D stages, each taking d steps, all messages are routed. We summarize this result below.

THEOREM 7.9.4 *Each computation cycle of a p-processor EREW PRAM can be simulated by a normal algorithm on a p-vertex hypercube in $O(D\log p+\log^2 p)$ steps, where D is the maximum number of processors accessing memory locations stored at a given vertex of the hypercube.*

This result can be improved to $O(\log p)$ [157] with a probabilistic algorithm that replicates each datum at each hypercube processor a fixed number of times.

Because the simulation described above of a EREW PRAM on a hypercube consists of a fixed number of normal steps and fully normal sequences of steps, $O(D\sqrt{p})$- and $O(Dp)$-time simulations of a PRAM on two-dimensional meshes and linear arrays follow. (See Problems 7.32 and 7.33.)

7.9.4 Circuits and the CREW PRAM

Algebraic and logic circuits can also be simulated on PRAMs, in particular the CREW PRAM. For simplicity we assign one processor to each vertex of a circuit (a gate). We also assume that each vertex has bounded fan-in, which for concreteness is assumed to be 2. We also reserve one memory location for each gate and one for each input variable. Each processor now alternates between reading values from its two inputs (concurrently with other processors, if necessary) and exclusively writing values to the location reserved for its value. Two steps are devoted to reading the values of gate inputs. Let $D_\Omega(f)$ be the depth of the circuit for a function f. After $2D_\Omega(f)$ steps the input values have propagated to the output gates, the values computed by them are correct and the computation is complete.

In Section 8.14 we show a stronger result, that CREW PRAMs and circuits are equivalent as language recognizers. We also explore the parallel computation thesis, which states that sequential space and parallel time are polynomially related. It follows that the PRAM and the logic circuit are both excellent models in terms of which to measure the minimal computation time required for a problem on a parallel machine. In Section 8.15 we exhibit complexity classes, that is, classes of languages defined in terms of the depth of circuits recognizing them.

7.10 The BSP and LogP Models

Bulk synchronous parallelism (BSP) extends the MIMD model to potentially different asynchronous programs running on the physical processors of a parallel computer. Its developers believe that the BSP model is both built on realistic assumptions and sufficiently simple to provide an attractive model for programming parallel computers. They expect it will play a

role similar to that of the RAM for serial computation, that is, that programs written for the BSP model can be translated into efficient code for a variety of parallel machines.

The BSP model explicitly assumes that a) computations are divided into *supersteps*, b) all processors are synchronized after each superstep, c) processors can send and receive messages to and from all other processors, d) message transmission is non-blocking (computation can resume after sending a message), and e) all messages are delivered by the end of a superstep. The important parameters of this model are p, the number of processors, s, the speed of each processor, l, the latency of the system, which is the number of processor steps to synchronize processors, and g, the additional number of processor steps per word to deliver a message. Here g measures the time per word to transmit a message between processors after the path between them has been set up; l measures the time to set up paths between processors and/or to synchronize all p processors. Each of these parameters must be appraised under "normal" computational and communication loads if the model is to provide useful estimates of the time to complete a task.

For the BSP model to be effective, it must be possible to keep the processors busy while waiting for communications to be completed. If the latency of the network is too high, this will not be possible. It will also not be possible if algorithms are not designed properly. For example, if all processors attempt to send messages to a single processor, network congestion will prevent the messages from being answered quickly. It has been shown that for many important problems data can be distributed and algorithms designed to make good use of the BSP model [347]. It should also be noted that the BSP model is not effective on problems that are not parallelizable, such as may be the case for **P**-complete problems (see Section 8.9).

Although for many problems and machines the BSP model is a good one, it does not take into account network congestion due to the number of messages in transit. The **LogP model** extends the BSP model by explicitly accounting for the overhead time (the o in LogP) to prepare a message for transmission. The model is also characterized by the parameters L, g, and P that have the same meaning as the parameters l, g, and p in the BSP model. The LogP and BSP models are about equally good at predicting algorithm performance.

Many other models have been proposed to capture one aspect or another of practical parallel computation. Chapter 11 discusses some of the parallel I/O issues.

Problems

PARALLEL COMPUTERS WITH MEMORY

7.1 Consider the design of a bus arbitration sequential circuit for a computer containing four CPUs. This circuit has four Boolean inputs and outputs, one per CPU. A CPU requesting bus access sets its input to 1 and waits until its output is set to 1, after which it puts its word and destination address on the bus. CPUs not requesting bus access set their bus arbitration input variable to 0.

At the beginning of each cycle the bus arbitration circuit reads the input variables and, if at least one of them has value 1, sets one output variable to 1. If all input variables are 0, it sets all output variables to 0.

Design two such arbitration circuits, one that grants priority to the lowest indexed input that is 1 and a second that grants priority alternately to the lowest and highest indexed input if more than one input variable is 1.

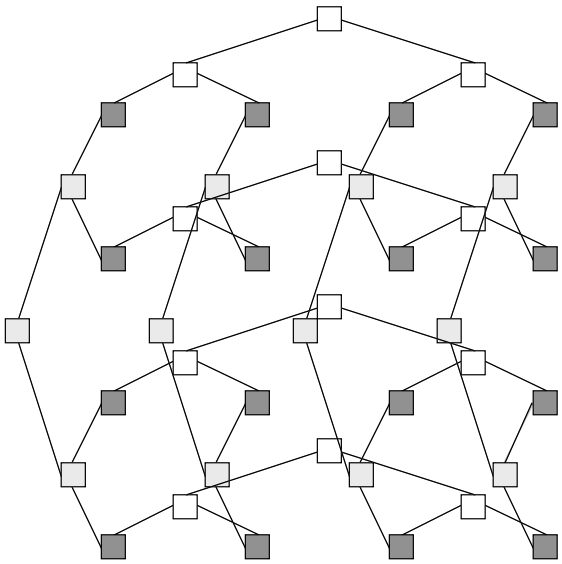

Figure 7.22 A four-by-four mesh-of-trees network.

7.2 Sketch a data-parallel program that operates on a sorted list of keys and finds the largest number of times that a key is repeated.

7.3 Sketch a data-parallel program to find the last record in a linked list where initially each record contains the address of the next item in the list (except for the last item, whose next address is *null*).

Hint: Assign one processor to each list item and assume that accesses to two or more distinct addresses can be done simultaneously.

7.4 The $n \times n$ mesh-of-trees network, $n = 2^r$, is formed from a $n \times n$ mesh by replacing each linear connection forming a row or column by a balanced binary tree. (See Fig. 7.22.) Let the entries of two $n \times n$ matrices be uniformly distributed on the vertices of original mesh. Give an efficient matrix multiplication algorithm on this network and determine its running time.

7.5 Identify problems that arise in a crossbar network when more than one source wishes to connect to the same destination. Describe how to insure that only one source is connected to one destination at the same time.

THE PERFORMANCE OF PARALLEL ALGORITHMS

7.6 Describe how you might apply Amdahl's Law to a data-parallel program to estimate its running time.

7.7 Consider the evaluation of the polynomial $p(x) = a_n x^n + x_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ on a p-processor shared-memory machine. Sketch an algorithm whose running time is $O(\frac{n}{p} + \log n)$ for this problem.

LINEAR ARRAYS

7.8 Generalize the example of Section 7.5.1 to show that the product of an $n \times n$ matrix and an n-vector can be realized in $3n - 1$ steps on a linear systolic array.

7.9 Show that every algorithm on a linear array to compute the product of an $n \times n$ matrix and an n-vector requires at least n steps. Assume that components of the matrix and vector enter cells individually.

7.10 Design an algorithm for a linear array of length $O(n)$ that convolves two sequences each of length n in $O(n)$ steps. Show that no substantially faster algorithm for such a linear array exists.

MULTIDIMENSIONAL ARRAYS

7.11 Show that at most $\sigma(d) = 2d^2 + 2d + 1$ cells are at most d edges away from any cell in a two-dimensional systolic array.

7.12 Derive an expression for the distance between vertices $(n_1, n_2, \ldots, n_d)$ and $(m_1, m_2, \ldots, m_d)$ in a d-dimensional toroidal mesh and determine the maximum distance between two such vertices.

7.13 Design efficient algorithms to multiply two $n \times n$ matrices on a $k \times k$ mesh, $k \leq n$.

HYPERCUBE-BASED MACHINES

7.14 Show that the vertices of the 2^d-input FFT graph can be numbered so that edges between levels correspond to swaps across the dimensions of a d-dimensional hypercube.

7.15 Show that the convolution function $f_{\text{conv}}^{(n,m)} : R^{n+m} \mapsto R^{n+m-1}$ over a commutative ring $\mathcal{R}$ can be implemented by a fully normal algorithm in time $O(\log n)$.

7.16 Prove that the unshuffle operation on a linear array of $n = 2^d$ cells can be done with $2^d - 1$ comparison/exchange steps.

7.17 Prove that the algorithm described in Section 7.7.4 to simulate a normal hypercube algorithm on a linear array of $n = 2^d$ elements correctly places into exchange locations elements whose indices differ by successive powers of 2.

7.18 Describe an efficient algorithm for a linear array that merges two sorted sequences of the same length.

7.19 Show that Batcher's sorting algorithm based on bitonic merging can be realized on an p-vertex hypercube by a normal algorithm in $O(\log^2 p)$ steps.

7.20 Show that Batcher's sorting algorithm based on bitonic merging can be realized on a linear array of $n = 2^d$ cells in $O(n)$ steps.

7.21 Show that Batcher's sorting algorithm based on bitonic merging can be realized on an $\sqrt{n} \times \sqrt{n}$ array in $O(\sqrt{n})$ steps.

7.22 Design an $O(\sqrt{n})$-step algorithm to implement an arbitrary permutation of n items placed one per cell of an $\sqrt{n} \times \sqrt{n}$ mesh.

7.23 Describe a normal algorithm to realize a prefix computation on a p-vertex hypercube in $O(\log p)$ steps.

7.24 Design an algorithm to perform a prefix computation on an $\sqrt{n} \times \sqrt{n}$ mesh in $3\sqrt{n}$ steps. Show that no other algorithm for this problem on this mesh has substantially better performance.

ROUTING IN NETWORKS

7.25 Give a complete description of a procedure to set up the switches in a Beneš network.

7.26 Show how to perform an arbitrary permutation on a linear array.

THE PRAM MODEL

7.27 a) Design an $O(1)$-step CRCW PRAM algorithm to find the maximum element in a list.

b) Design an $O(\log \log n)$-step CRCW PRAM algorithm to find the maximum element in a list that uses $O(n)$ processors.

Hint: Construct a tree in which the root and every other vertex has a number of immediate descendants that is about equal to the square root of the number of leaves that are its descendants.

7.28 The goal of the **list-ranking problem** is to assign a rank to each record in a linked list; the rank of a record is its position relative to the last element in the list where the last element has rank zero. Each record has two fields, one for its rank and another for the address of its successor record. The address field of the last record contains its own address.

Describe an efficient p-processor EREW PRAM algorithm to solve the list-ranking problem for a list of p items stored one per location in the common memory.

Hint: Use **pointer doubling** in which each address is replaced by the address of its current successor.

7.29 Consider an n-vertex directed graph in which each vertex knows the address of its parent and the roots have themselves as parents. Under the assumption that each vertex is placed in a unique cell in a common PRAM memory, show that the roots can be found in $O(\log n)$ steps.

7.30 Design an efficient PRAM algorithm to find the item in a list that occurs most often.

7.31 Figure 7.23 shows two trees containing one and three copies of a computational element, respectively. This element accepts three inputs and produces three outputs using $\odot$, an associative operator. Tree (a) accepts a, b, and c as input and produces a, $a \odot b$, and $b \odot c$ as output. Tree (b) accepts a, b, c, d, and e as input and produces a, $a \odot b$, $a \odot b \odot c$, $a \odot b \odot c \odot d$, and $b \odot c \odot d \odot e$ as output. If the input and output at the root of the trees are combined with $\odot$, the output of each tree is the prefix computation on its inputs.

Generalize the constructions of Fig. 7.23 to produce a circuit for the prefix function on n inputs, n arbitrary. Give a convincing argument that your construction is correct and derive good upper bounds on the size and depth of your circuit. Show that to within multiplicative factors your construction has minimal size and depth.

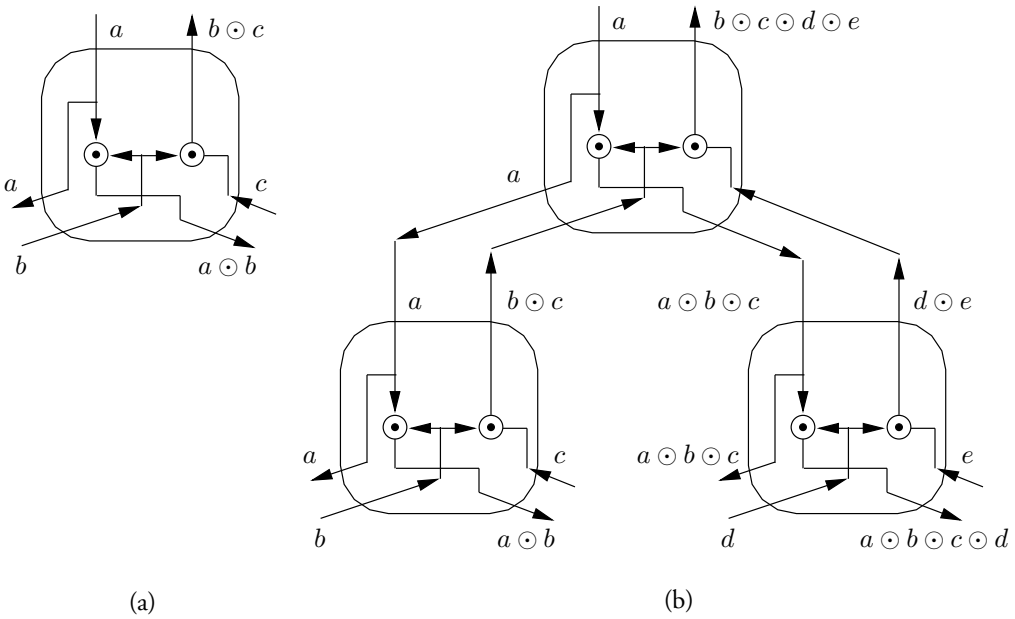

Figure 7.23 Components of an efficient prefix circuit.

7.32 Show that each computation cycle of a p-processor EREW PRAM can be simulated on a $\sqrt{p} \times \sqrt{p}$ mesh in $O(D\sqrt{p})$ steps, where D is the maximum number of processors accessing memory locations stored at a given vertex of the mesh.

7.33 Show that each computation cycle of a p-processor EREW PRAM can be simulated on a p-processor linear array in $O(Dp)$ steps, where D is the maximum number of processors accessing memory locations stored at a given vertex of the array.

THE BSP AND LOGP MODELS

7.34 Design an algorithm for the p-processor BSP and/or LogP models to multiply two $n \times n$ matrices when each matrix entry occurs once and entries are uniformly distributed over the p processors. Given the parameters of the models, determine for which values of n your algorithm is efficient.

Hint: The performance of your algorithm will be dependent on the initial placement of data.

7.35 Design an algorithm for the p-processor BSP and/or LogP models for the segmented prefix function. Given the parameters of the models, determine for which values of n your algorithm is efficient.

Chapter Notes

A discussion of parallel algorithms and architectures up to about 1980 can be found in the book by Hockney and Jesshope [134]. A number of recent textbooks provide extensive coverage of

parallel algorithms and architectures. They include the books by Akl [16], Bertsekas and Tsitsiklis [38], Gibbons and Spirakis [112], JáJá [147], Leighton [191], Quinn [264], and Reif [276]. In addition, the survey article by Karp and Ramachandran [160] gives an overview of parallel algorithmic methods. References to results on circuit complexity can be found in Chapters 2, 6, and 9.

Flynn introduced the taxonomy of parallel computers that carries his name [101]. The data-parallel style of computing was anticipated in the APL [145] and FP programming languages [26] as well as by Preparata and Vuillemin [261] in their study of parallel algorithms for networked machines. It was developed as the style of choice for programming the Connection Machine [132]. (See also the books by Hatcher and Quinn [128] and Blelloch [45] on data-parallel computing.) The simulation of the MIMD computer by a SIMD one given in Section 7.3.1 is due to Wloka [364].

Amdahl's Law [21] and Brent's principle [58] are widely cited; the latter is used extensively to design efficient parallel algorithms.

Systolic algorithms for convolution, matrix multiplication, and the fast Fourier transform are given by Kung and Leiserson [179] (see also [180]). Odd-even transposition sort is described by Knuth [169]. The lower bound on the time to multiply two matrices given in Theorem 7.5.3 is due to Gentleman [111]. The shuffle network was introduced by Stone [317].

Preparata and Vuillemin [261] give normal algorithms for a variety of problems (including that for shifting in Section 7.7.3) and introduce the cube-connected cycles machine. They also give embeddings of fully normal algorithms into linear arrays and meshes. Dekel, Nassimi, and Sahni [85] developed the fast algorithm for matrix multiplication on the hypercube described in Section 7.7.7.

Batcher [29] introduced odd-even and bitonic sorting methods and noted that they could be used for routing messages in networks. Beneš [36] is the author of the Beneš permutation network.

Variants of the PRAM were introduced by Fortune and Wyllie [102], Goldschlager [117], Savitch and Stimson [297] as generalizations of the idealized RAM model of Cook and Reckhow [77]. The method given in Theorem 7.9.3 to simulate a CRCW PRAM on an EREW PRAM is due to Eckstein [94] and Vishkin [352]. Simulations of PRAMs on networked computers have been developed by Mehlhorn and Vishkin [220], Upfal [339], Upfal and Wigderson [340], Karlin and Upfal [157], Alt, Hagerup, Mehlhorn, and Preparata [19], and Ranade [266]. Cypher and Plaxton [84] have developed a deterministic $O(\log p \log\log p)$-step sorting algorithm for the hypercube. However, it is superior to Batcher's algorithm only for very large and impractical values of p.

The bulk synchronous parallel (BSP) model [347] has been proposed as a bridging model between the needs of programmers and parallel machines. The LogP model [83] is offered as a more realistic variant of the BSP model. Juurlink and Wijshoff [153] and Bilardi, Herley, Pietracaprina, Pucci, and Spirakis [39] report empirical evidence that the BSP and LogP models are about equally good as predictors of performance on real parallel computers.