Introduction to Parallel Computing

Design and Analysis of Algorithms

Vipin Kumar
Ananth Grama
Anshul Gupta
George Karypis

Read all of
Chapter 2
except 2.5

The Benjamin/Cummings Publishing Company, Inc.
Redwood City, California ▪ Fort Collins, Colorado ▪ Menlo Park, California
Reading, Massachusetts ▪ New York ▪ Don Mills, Ontario ▪ Wokingham, U.K.
Amsterdam ▪ Bonn ▪ Sydney ▪ Singapore ▪ Tokyo ▪ Madrid ▪ San Juan

CHAPTER 2

Models of Parallel Computers

Traditional sequential computers are based on the model introduced by John von Neumann. As shown in Figure 2.1, this model consists of a central processing unit (CPU) and memory. This computational model takes a single sequence of instructions and operates on a single sequence of data. Computers of this type are often referred to as ***single instruction stream, single data stream*** (SISD) computers.

The speed of an SISD computer is limited by two factors: the execution rate of instructions and the speed at which information is exchanged between memory and the CPU. The latter can be increased by increasing the number of channels on which data can be accessed simultaneously. This is done by dividing memory into a number of banks, each of which is accessed independently (Figure 2.1(b)). This is called ***memory interleaving***.

Another way to increase the rate of information exchange between the CPU and memory is to use a relatively small and very fast memory to act as a buffer to the larger, slower primary memory. This fast memory is called ***cache memory*** (Figure 2.1(c)). It is possible to fetch a block of data from main memory into the cache, from which data can be exchanged with the CPU at much higher transfer rates. Cache memory uses the principle that if a word is accessed from a part of the memory, it is likely that subsequent memory accesses will be to words in the neighborhood of this memory location.

The rate of execution of instructions can also be increased by overlapping the execution of an instruction with the operation of fetching the next instruction to be executed. Thus, while the CPU is busy executing the current instruction, the next instruction is brought from memory into the instruction queue. This technique is called ***instruction pipelining***. In a related technique called ***execution pipelining***, multiple instructions are allowed to be in various stages of execution in functional units such as multipliers and adders (Figure 2.1(d)).

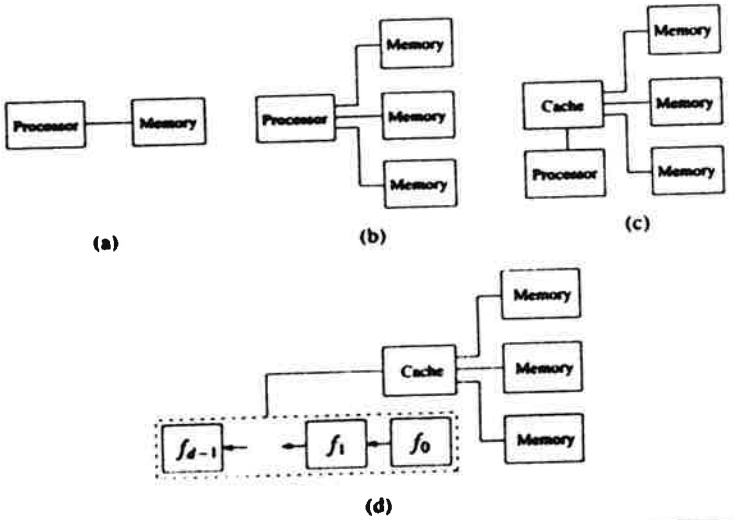

Figure 2.1 The evolution of a typical sequential computer: (a) a simple sequential computer; (b) a sequential computer with memory interleaving; (c) a sequential computer with memory interleaving and cache; and (d) a pipelined processor with d stages.

Memory interleaving, cache memory, and pipelining are now commonly used in high-performance SISD computers; however, they all have limitations. Memory interleaving and, to some extent, pipelining are useful only if a small set of operations is performed on large arrays of data. Cache memories do increase processor-memory bandwidth, but their speed is still limited by hardware technology. An alternate way to speed up the rate of instruction execution is to use multiple CPUs and memory units interconnected in some fashion. The processing rate of such a system grows when the number of CPUs and memory units is increased.

2.1 A Taxonomy of Parallel Architectures

There are many ways in which parallel computers can be constructed. These computers differ along various dimensions such as control mechanism, address-space organization, interconnection network, and granularity of processors.

2.1.1 Control Mechanism

Processing units in parallel computers either operate under the centralized control of a single control unit or work independently. In architectures referred to as ***single instruction***

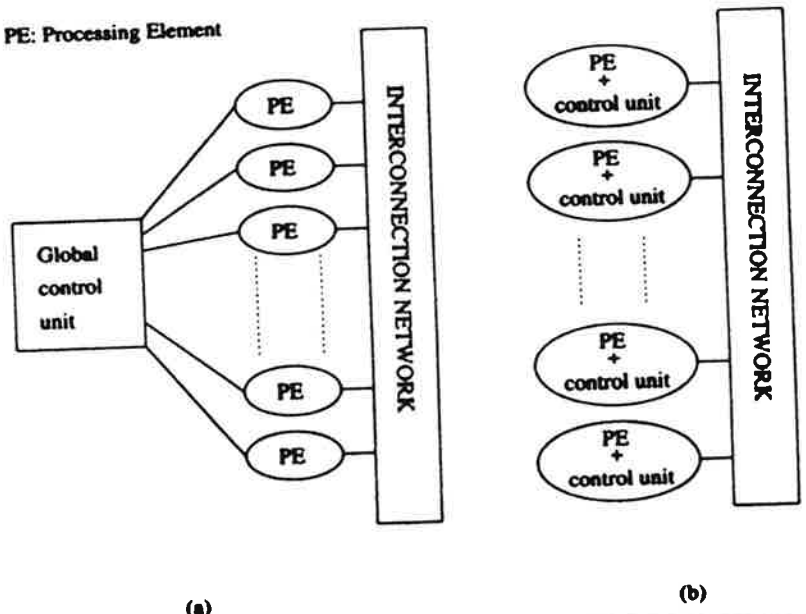

Figure 2.2 A typical SIMD architecture (a) and a typical MIMD architecture (b).

stream, multiple data stream (SIMD), a single control unit dispatches instructions to each processing unit. Figure 2.2(a) illustrates a typical SIMD architecture. In an SIMD parallel computer, the same instruction is executed synchronously by all processing units. Processing units can be selectively switched off during an instruction cycle. Examples of SIMD parallel computers include the Illiac IV, MPP, DAP, CM-2, MasPar MP-1, and MasPar MP-2.

Computers in which each processor is capable of executing a different program independent of the other processors are called ***multiple instruction stream, multiple data stream*** (MIMD) computers. Figure 2.2(b) depicts a typical MIMD computer. Examples of MIMD computers include the Cosmic Cube, nCUBE 2, iPSC, Symmetry, FX-8, FX-2800, TC-2000, CM-5, KSR-1, and Paragon XP/S.

SIMD computers require less hardware than MIMD computers because they have only one global control unit. Furthermore, SIMD computers require less memory because only one copy of the program needs to be stored. In contrast, MIMD computers store the program and operating system at each processor. SIMD computers are naturally suited for ***data-parallel programs***; that is, programs in which the same set of instructions are executed on a large data set. Furthermore, SIMD computers require less startup time (Section 2.7) for communicating with neighboring processors. This is because the communication of a

word of data is just like a register transfer (due to the presence of a global clock) with the destination register in the neighboring processor.

A drawback of SIMD computers is that different processors cannot execute different instructions in the same clock cycle. For instance, in a conditional statement, the code for each condition must be executed sequentially. This is illustrated in Figure 2.3. The conditional statement in Figure 2.3(a) is executed in two steps. In the first step, all processors that have B equal to zero execute the instruction $C = A$. All other processors are idle. In the second step, the 'else' part of the instruction ($C = A/B$) is executed. The processors that were active in the first step now become idle. Data-parallel programs in which significant parts of the computation are contained in conditional statements are therefore better suited to MIMD computers than to SIMD computers.

Individual processors in an MIMD computer are more complex, because each processor has its own control unit. It may seem that the cost of each processor must be higher than the cost of a SIMD processor. However, it is possible to use general-purpose microprocessors as processing units in MIMD computers. In contrast, the CPU used in SIMD computers has to be specially designed. Hence, due to the economy of scale, processors in MIMD computers may be both cheaper and more powerful than processors in SIMD computers.

SIMD computers offer automatic synchronization among processors after each instruction execution cycle. Hence, SIMD computers are better suited to parallel programs that require frequent synchronization. Many MIMD computers have extra hardware to provide fast synchronization, which enables them to operate in SIMD mode as well. Examples of such computers are the DADO and CM-5.

2.1.2 Address-Space Organization

Solving a problem on an ensemble of processors requires interaction among processors. The ***message-passing*** and ***shared-address-space*** architectures provide two different means of processor interaction.

Message-Passing Architecture In a message-passing architecture, processors are connected using a message-passing interconnection network. Each processor has its own memory called the ***local*** or ***private memory***, which is accessible only to that processor. Processors can interact only by passing messages. This architecture is also referred to as a ***distributed-memory*** or ***private-memory*** architecture. Figure 2.4 shows the architecture of a typical message-passing architecture. MIMD message-passing computers are commonly referred to as ***multicomputers***. Examples of message-passing parallel computers include the Cosmic Cube, Paragon XP/S, iPSC, CM-5, and nCUBE 2.

Shared-Address-Space Architecture The shared-address-space architecture provides hardware support for read and write access by all processors to a shared address space. Processors interact by modifying data objects stored in the shared address space. MIMD shared-address-space computers are often referred to as ***multiprocessors***.

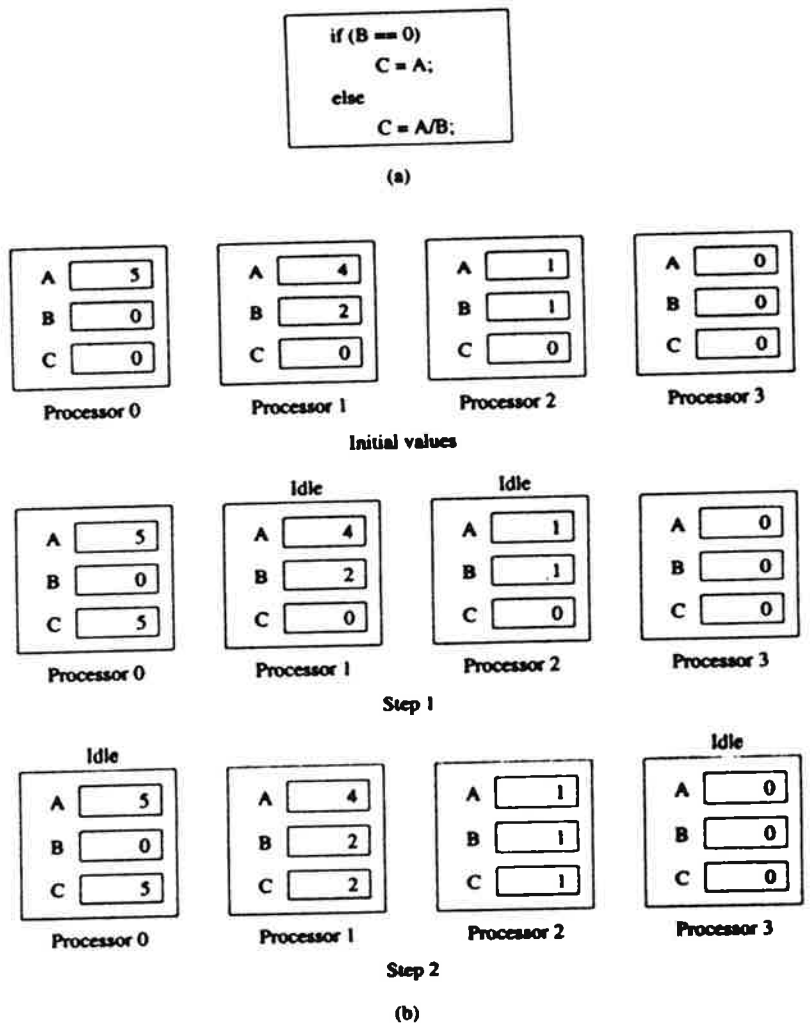

Figure 2.3 Executing a conditional statement on an SIMD computer with four processors: (a) The conditional statement; (b) The execution of the statement in two steps.

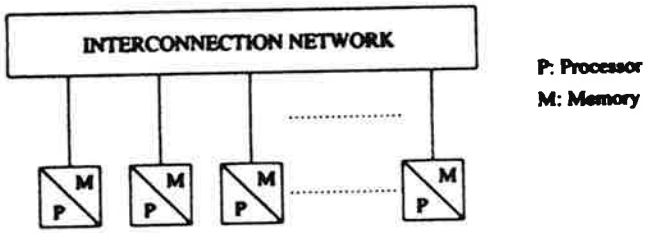

Figure 2.4 A typical message-passing architecture.

Early shared-address-space computers contain a shared memory (possibly organized into banks) that is equally accessible to all processors through an interconnection network (Figure 2.5(a)). These architectures are called *shared-memory* parallel computers. Examples of such computers are the C.mmp and the NYU Ultracomputer. A major drawback of these architectures is that the bandwidth of the interconnection network must be substantial to ensure good performance. This is because, in each instruction cycle, every processor may need to access a word from the shared memory through the interconnection network. Furthermore, memory access through the interconnection network can be slow, since a read or write request may have to pass through multiple stages in the network. Both of these problems can seriously degrade the performance of a shared-memory system.

One way to alleviate these drawbacks is to provide each processor with a local memory as shown in Figure 2.5(b). This memory stores the program being executed on the processor and any non-shared data structures. Global data structures are stored in the shared

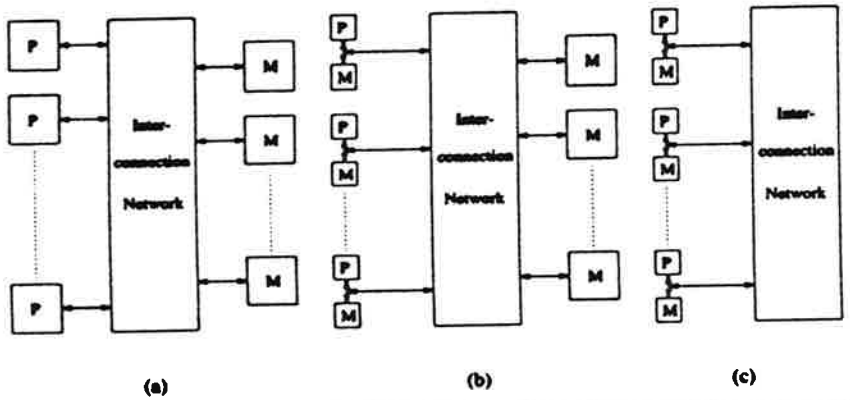

Figure 2.5 Typical shared-address-space architectures: (a) Uniform-memory-access shared-address-space computer; (b) Non-uniform-memory-access shared-address-space computer with local and global memories; (c) Non-uniform-memory-access shared-address-space computer with local memory only.

memory. This scheme eliminates repeated memory references across the interconnection network and thus improves performance.

The local memory concept can be extended to eliminate physically shared memory entirely (Figure 2.5(c)). Memory references to words in other processors' local memories are then mapped by the hardware to the appropriate processors. In such an architecture, local memory access time is much smaller than remote memory access time.

Based on the amount of time a processor takes to access local and global memory, shared-address-space computers are classified into two categories. If the time taken by a processor to access any memory word in the system is identical, the computer is classified as a *uniform memory access* (UMA) computer. On the other hand, if the time to access a remote memory bank is longer than the time to access a local one, the computer is called a *nonuniform memory access* (NUMA) computer. Figure 2.5(a) represents a UMA computer, whereas Figures 2.5(b) and (c) represent NUMA computers.

Most shared-address-space computers also have a local cache at each processor to increase their effective processor-memory bandwidth. As in sequential computers, a cache provides faster access to the data contained in the local memory. The cache can also be used to provide fast access to remotely-located shared data. Whenever a processor needs data that is located in a non-local memory, it is copied into the local cache. Subsequent access to this data is very fast. The use of a cache introduces the problem of *cache coherence*. This problem occurs when a processor modifies a shared variable in its cache. After this modification, different processors have different values of the variable, unless copies of the variable in the other caches are simultaneously invalidated or updated. Several mechanisms have been developed for handling the cache coherence problem.

Note that the NUMA architecture is similar to a message-passing architecture—memory is physically distributed in both. The major difference between them is that a NUMA architecture provides hardware support for read and write access to other processors' memories, whereas in a message-passing architecture, remote access must be emulated by explicit message passing. For historical reasons, the NUMA architecture is often referred to as a shared-memory architecture due to its shared address-space. Most recent shared-address-space computers are NUMA computers. Examples of such computers include the TC-2000, KSR-1, and Stanford Dash. In the KSR-1, each processor contains only cache and no local memory. Such an architecture is referred to as *cache-only memory access* (COMA) architecture.

It is easy to emulate a message-passing architecture containing p processors on a shared-address-space computer with an identical number of processors. We do this by partitioning the shared address space into p disjoint parts and assigning one such partition exclusively to each processor. A processor sends a message to another processor by writing into the other processor's partition of memory. However, emulating a shared-address-space architecture on a message-passing computer is costly, since accessing another processor's memory requires sending and receiving messages. Hence, shared-address-space computers provide greater flexibility in programming. Furthermore, some problems require rapid access by all processors to large data structures that may be changing dynamically. Such

access is better supported by shared-address-space architectures. However, the hardware needed to provide a shared address space tends to be more expensive than that for message passing.

2.1.3 Interconnection Networks

Shared-address-space computers and message-passing computers can be constructed by connecting processors and memory units using a variety of interconnection networks. Interconnection networks can be classified as ***static*** or ***dynamic***. Static networks (Section 2.4) consist of point-to-point communication links among processors and are also referred to as ***direct*** networks. Static networks are typically used to construct message-passing computers. Dynamic networks (Section 2.3) are built using switches and communication links. Communication links are connected to one another dynamically by the switching elements to establish paths among processors and memory banks. Dynamic networks are referred to as ***indirect*** networks and are normally used to construct shared-address-space computers.

2.1.4 Processor Granularity

A parallel computer may be composed of a small number of very powerful processors or a large number of relatively less powerful processors. Processors belonging to the former class are called ***coarse-grain*** computers, and those belonging to the latter are called ***fine-grain*** computers. Machines along the entire spectrum are now commercially available. Coarse grain computers such as the Cray Y-MP offer a small number of processors (8 to 16), each capable of several Gflops (one Gflops equals 10^9 floating-point operations per second). In contrast, fine-grain computers such as the CM-2, MasPar MP-1, and MasPar MP-2 offer a large number of relatively slow processors (for example, the CM-2 contains up to 65,536 one-bit processors, and the MasPar MP-1 contains up to 16,384 four-bit processors). Between these extremes are ***medium-grain*** computers such as the CM-5, nCUBE 2, and Paragon XP/S, containing up to a few thousand processors, each delivering workstation-class performance.

Individual processors in coarse-grain computers are considerably more expensive than those in fine- and medium-grain computers. The reason for this is that the fast processors used in coarse-grain computers are not produced on a large scale. Furthermore, these processors require expensive fabrication techniques. Medium-grain computers, however, are often constructed from inexpensive off-the-shelf hardware.

Different applications are suited to coarse-, medium-, or fine-grain computers to varying degrees. Many applications have only a limited amount of concurrency. Such applications cannot make effective use of a large number of less powerful processors, and are best suited to coarse-grain computers. Fine-grain computers, however, are more cost effective for applications with a high degree of concurrency. Thus, we must make a tradeoff between the cost and the utility of the computer when choosing processor granularity.

The granularity of a parallel computer can be defined as the ratio of the time required for a basic communication operation to the time required for a basic computation. Parallel computers for which this ratio is small are suitable for algorithms requiring frequent communication; that is, algorithms in which the grain size of the computation (before a communication is required) is small. Since such algorithms contain fine-grain parallelism, these parallel computers are often called fine-grain computers. In contrast, parallel computers for which this ratio is large are suited to algorithms that do not require frequent communication. These computers are referred to as coarse-grain computers. According to this criterion, multicomputers such as the nCUBE 2 and Paragon XP/S are coarse-grain computers, whereas multiprocessors such as the C.mmp, TC-2000, and KSR-1 are fine-grain computers.

2.2 An Idealized Parallel Computer

In this section, we define a theoretical model of computation based on shared-memory computers. This model is referred to as a ***parallel random access machine (PRAM)***. Formally, a PRAM consists of p processors and a global memory of unbounded size that is uniformly accessible to all processors. Thus, all processors access the same address space. Processors share a common clock but may execute different instructions in each cycle. PRAM models are therefore synchronous shared-memory MIMD computers. A PRAM model is idealized in the sense that it is a natural extension of the sequential model of computation and provides a means of interaction between processors at no cost.

In any shared-memory parallel computer, more than one processor can try to read from or write into the same memory location simultaneously. The PRAM model can be divided into four subclasses, based on how simultaneous memory accesses are handled.

(1) ***Exclusive-read, exclusive-write (EREW) PRAM.*** In this class, access to a memory location is exclusive. No concurrent read or write operations are allowed. This is the weakest PRAM model, affording minimum concurrency in memory access.

(2) ***Concurrent-read, exclusive-write (CREW) PRAM.*** In this class, multiple read accesses to a memory location are allowed. However, multiple write accesses to a memory location are serialized.

(3) ***Exclusive-read, concurrent-write (ERCW) PRAM.*** Multiple write accesses are allowed to a memory location, but multiple read accesses are serialized.

(4) ***Concurrent-read, concurrent-write (CRCW) PRAM.*** This class allows multiple read and write accesses to a common memory location. This is the most powerful PRAM model. However, it is possible to simulate CRCW on an EREW model (Problem 2.1).

Allowing concurrent read access does not create any semantic discrepancies in the program. However, concurrent write access to a memory location requires arbitration. Several protocols are used to resolve concurrent writes. The most frequently used protocols are as follows:

- ***Common***, in which the concurrent write is allowed if all the values that the processors are attempting to write are identical.
- ***Arbitrary***, in which an arbitrary processor is allowed to proceed with the write operation and the rest fail.
- ***Priority***, in which all processors are organized into a predefined prioritized list, and the processor with the highest priority succeeds and the rest fail.
- ***Sum***, in which the sum of all the quantities is written (the sum-based write conflict resolution model can be extended to any associative operator defined on the quantities being written).

2.3 Dynamic Interconnection Networks

Consider the implementation of an EREW PRAM as a shared-memory computer with p processors and a global memory of m words. The processors are connected to the memory through a set of switching elements. These switching elements determine the memory word being accessed by each processor. In an EREW PRAM, each of the p processors in the ensemble can access any of the memory words, provided that a word is not accessed by more than one processor simultaneously. To ensure such connectivity, the total number of switching elements must be $\Theta(mp)$. For a reasonable memory size, constructing a switching network of this complexity is very expensive. Thus, PRAM models of computation are impossible to realize in practice.

One way to reduce the complexity of the switching network is to reduce the value of m by organizing the memory into banks. A processor switches between banks and not between individual memory words. If the total memory m was organized into b banks, then each of the p processors needs to switch between b banks. This model is only a weak approximation of the EREW PRAM model, because if a processor accesses even one word in a memory bank, no other processor can access any word in the same memory bank.

We now describe some important dynamic interconnection networks used in practical shared-memory architectures.

2.3.1 Crossbar Switching Networks

A simple way to connect p processors to b memory banks is to use a crossbar switch. A crossbar switch employs a grid of switching elements as shown in Figure 2.6. The crossbar switching network is a nonblocking network in the sense that the connection of a processor to a memory bank does not block the connection of any other processor to any other memory bank. Normally, b is greater than or equal to p so that each processor has at least one memory bank to access. If b approaches the size of the memory (that is, each memory bank has one word), then the crossbar simulates an EREW PRAM.

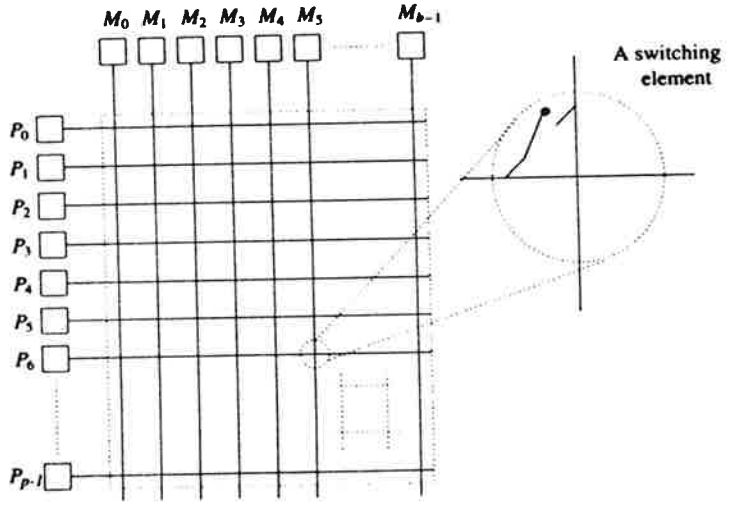

Figure 2.6 A completely nonblocking crossbar switch connecting p processors to b memory banks.

The total number of switching elements required to implement such a network is $\Theta(pb)$. It is reasonable to assume that the number of memory banks b is at least p; otherwise, at any given time, there will be some processors that will be unable to access any memory bank. Therefore, as the value of p is increased, the complexity of the switching network grows as $\Omega(p^2)$. Hence, the switching network becomes practically unrealizable as the number of processors becomes very large. Consequently, crossbar networks are not very scalable in terms of cost.

Several parallel processors based on the crossbar switch have been constructed. The Cray Y-MP and Fujitsu VPP 500 are recent examples of such computers. The VPP 500 uses a 224 × 224 crossbar network to connect 222 processors and 2 control processors. Implementing a crossbar switch of this magnitude is a task requiring considerable ingenuity.

2.3.2 Bus-Based Networks

In a bus-based network, processors are connected to global memory by means of a common data path called a ***bus***. Such a system is very simple to construct. Figure 2.7(a) illustrates a typical bus architecture. Whenever a processor accesses global memory, that processor generates a request over the bus. The data is then fetched from memory over the same bus.

Given its simplicity of construction and ability to provide uniform access to shared memory, this network appears very attractive. However, the bus can carry only a limited amount of data between the memory and the processors. If we increase the number of processors, each processor spends an increasing amount of time waiting for memory access

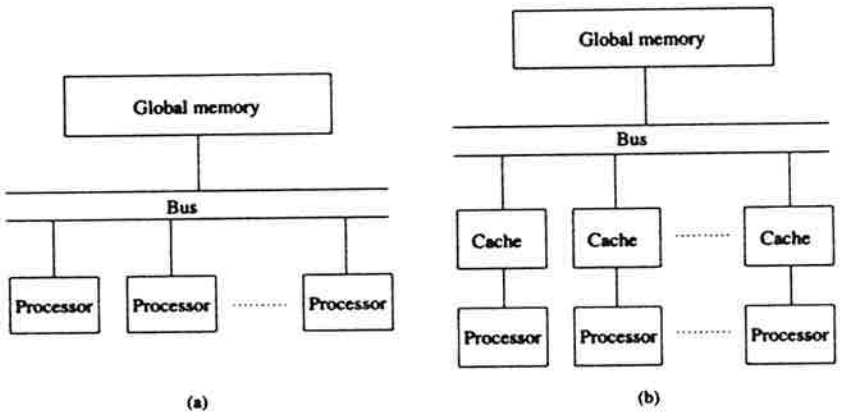

Figure 2.7 A typical bus-based architecture with no cache (a) and with cache memory at each processor (b).

while the bus is in use by other processors. Consequently, the performance of bus-based networks saturates at a small number of processors.

One way to alleviate the bus bottleneck is to provide each processor with a local cache memory, as shown in Figure 2.7(b). In a typical computation, when a reference is made to a memory location, subsequent references are likely to be made to memory locations in the neighborhood of this location. Due to this locality of reference of data and instructions, once a block of data is fetched into a processor's cache memory, subsequent references will likely be to memory words in the cache. In the case of a cache miss (that is, when the word accessed is not in the cache), a block of data containing the required word is brought from the global memory across the shared bus into the local cache.

Local cache memory thus reduces the total number of accesses to global memory. For systems with local cache memory, the bottleneck due to the limited data transmission rate of the bus manifests itself at a higher number of processors. However, replicating data this way leads to the cache coherence problems discussed in Section 2.1.2. Several cache coherence techniques have been developed for bus-based parallel computers containing a small number of processors.

Some of the commercially successful bus-based shared-address-space parallel computers are the Symmetry and Multimax.

2.3.3 Multistage Interconnection Networks

The crossbar interconnection network is scalable in terms of performance but unscalable in terms of cost. Conversely, the shared bus network is scalable in terms of cost but unscalable in terms of performance. An intermediate class of networks called ***multistage interconnection networks*** lies between these two extremes. It is more scalable than the bus in terms of performance and more scalable than the crossbar in terms of cost.

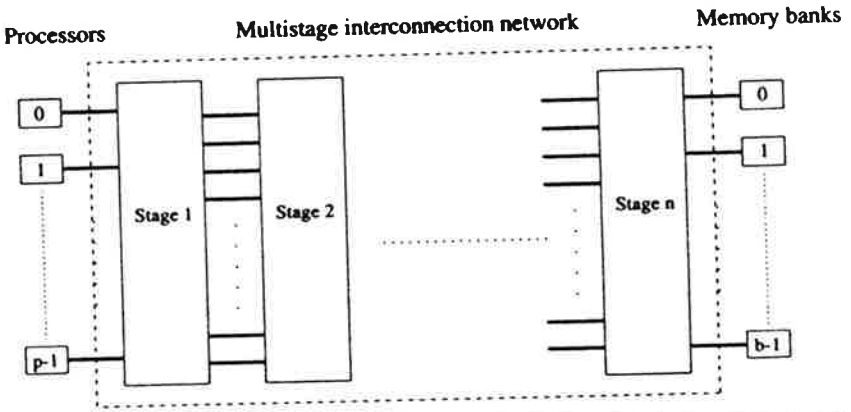

Figure 2.8 The schematic of a typical multistage interconnection network.

Figure 2.9(a) illustrates the cost of crossbar, multistage, and bus-based parallel computers and Figure 2.9(b) illustrates the performance characteristics of these computers in terms of total communication capacity.

The general schematic of a multistage network consisting of p processors and b memory banks is shown in Figure 2.8. A commonly used multistage connection network is the ***omega network***.This network consists of $\log p$ stages (where p is the number of processors and also the number of memory banks). Each stage of the omega network consists of an interconnection pattern that connects p inputs and p outputs; a link exists

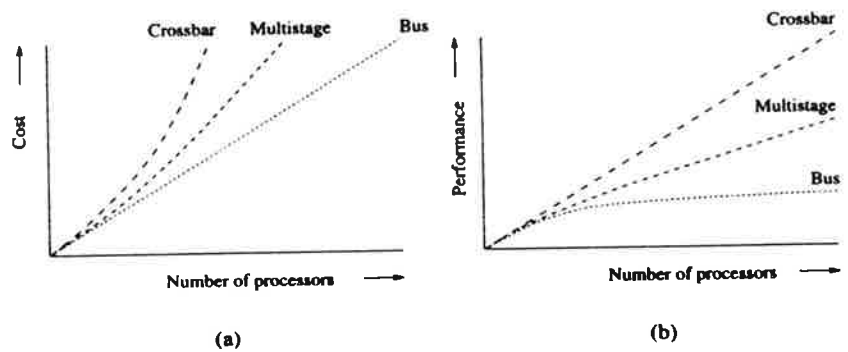

Figure 2.9 (a) Cost versus number of processors for interconnection networks based on bus, multistage, and crossbar connected networks; (b) Performance versus number of processors for the three networks.

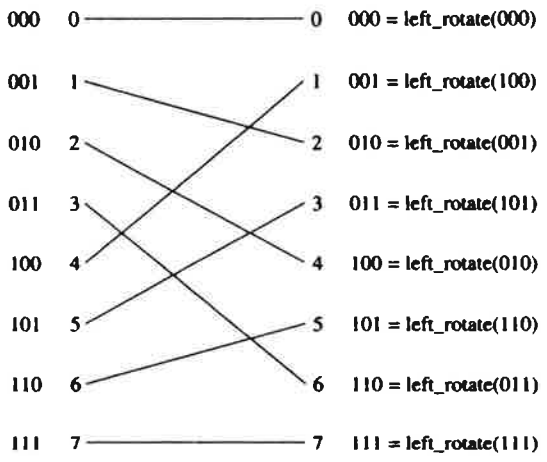

Figure 2.10 A perfect shuffle interconnection for eight inputs and outputs.

between input i and output j if the following is true:

$$j = \begin{cases} 2i, & 0 \leq i \leq p/2 - 1 \\ 2i + 1 - p, & p/2 \leq i \leq p - 1 \end{cases} \tag{2.1}$$

Equation 2.1 represents a left-rotation operation on the binary representation of i to obtain j. This interconnection pattern is called a ***perfect shuffle***. Figure 2.10 shows a perfect shuffle interconnection pattern for eight inputs and outputs. In each stage of an omega network, a perfect shuffle interconnection pattern feeds into a set of $p/2$ switching elements. Each switch is in one of two connection modes. In one mode, the inputs are sent straight through to the outputs, as shown in Figure 2.11(a). This is called the ***pass-through*** connection. In the other mode, the inputs to the switching element are crossed over and then sent out, as shown in Figure 2.11(b). This is called the ***cross-over*** connection. An omega network has $p/2 \times \log p$ switching elements, and the cost of such a network grows as $\Theta(p \log p)$. Note that this cost is less than the $\Theta(p^2)$ cost of a complete crossbar switch.

Figure 2.12 shows an omega network for eight processors. Processors form the input nodes of the network and memory banks form the output nodes. Routing messages in an omega network is accomplished by using a simple scheme. Let s and t be the binary representations of the source and destination of the message. The message traverses the link to the first switching element. If the most significant bits of s and t are the same, then the message is routed in pass-through mode by the switch. If these bits are different, then the message is routed through in crossover mode. This scheme is repeated at the next switching stage using the next most significant bit. Traversing $\log p$ stages uses all $\log p$ bits in the binary representations of s and t.

Figure 2.11 Two switching configurations of the 2×2 switch: (a) Pass-through; (b) Cross-over.

Figure 2.13 shows message routing over an eight-processor omega network from processor two (010) to seven (111) and from processor six (110) to four (100). This figure illustrates an important property of this network. When processor two (010) is communicating with processor seven (111), it blocks the path from processor six (110) to four (100). The communication link AB is used by both communication paths. Thus, in an omega network, access to a memory bank by a processor may disallow access to another memory bank by another processor. Networks with this property are referred to as ***blocking networks***.

Parallel computers based on the omega network include the BBN Butterfly, IBM RP-3, and NYU Ultracomputer.

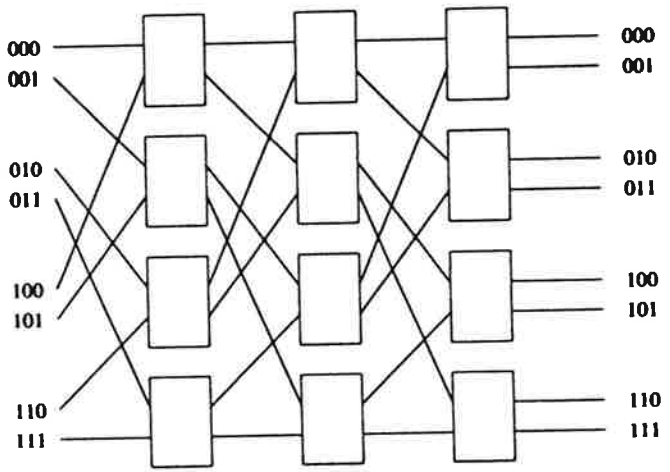

Figure 2.12 A complete omega network connecting eight inputs and eight outputs .

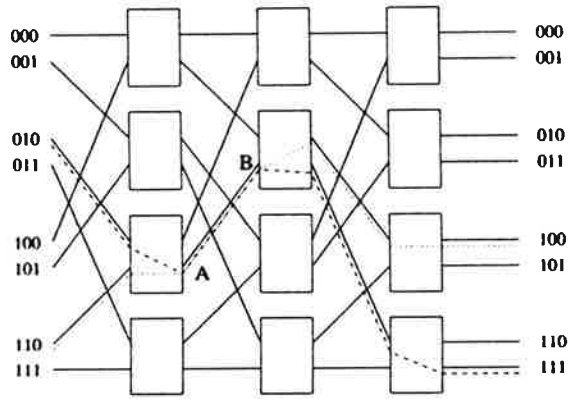

Figure 2.13 An example of blocking in omega network: one of the messages (010 to 111 or 110 to 100) is blocked at link AB.

2.4 Static Interconnection Networks

Message-passing architectures typically use static interconnection networks to connect processors. In this section, we discuss some important static interconnection networks and their properties.

2.4.1 Types of Static Interconnection Networks

Completely-Connected Network In a ***completely-connected network***, each processor has a direct communication link to every other processor in the network. Figure 2.14(a)

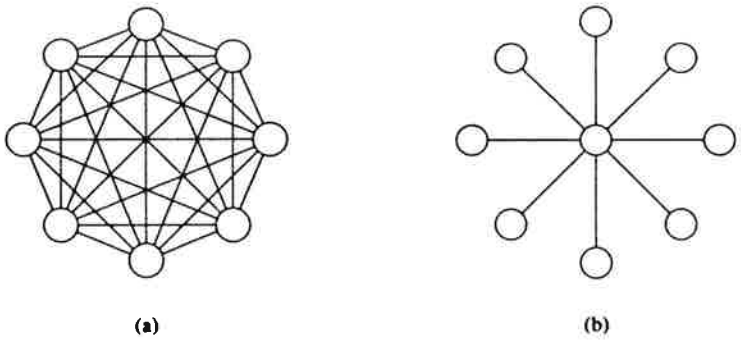

Figure 2.14 A completely-connected network of eight processors (a), and a star-connected network of nine processors (b).

Figure 2.15 A four-processor linear array (a) and a four-processor ring (b).

illustrates a completely-connected network of eight processors. This network is ideal in the sense that a processor can send a message to another processor in a single step, since a communication link exists between them. Completely-connected networks are the static counterparts of crossbar switching networks, since in both networks, the communication between any input/output pair does not block communication between any other pair. However, completely-connected networks can support communications over multiple channels originating at the same processor, whereas crossbar switches cannot.

Star-Connected Network In a ***star-connected network***, one processor acts as the central processor. Every other processor has a communication link connecting it to this processor. Figure 2.14(b) shows a star-connected network of nine processors. The star-connected network is similar to bus-based networks. Communication between any pair of processors is routed through the central processor, just as the shared bus forms the medium for all communication in a bus-based network. The central processor is the bottleneck in the star topology.

A network of workstations connected using an ethernet can be used as a parallel computer. The ethernet forms a common medium for communicating messages between processors. Such a network displays performance characteristics similar to star-connected and bus-based parallel computers.

Linear Array and Ring A simple way to connect processors is illustrated in Figure 2.15(a). Each processor in this network (except the processors at the ends) has a direct communication link to two other processors. Such an interconnection network is called a ***linear array***. A wraparound connection is often provided between the processors at the ends. A linear array with a wraparound connection is referred to as a ***ring***. Figure 2.15(b) shows a ring of four processors. One way of communicating a message between processors is by repeatedly passing it to the processor immediately to the right (or left, depending on which direction yields a shorter path) until it reaches its destination. Parallel computers using a ring network include the ZMOB and CDC Cyberplus.

Mesh Network The ***two-dimensional mesh*** is an extension of the linear array to two dimensions. In a two-dimensional mesh, each processor has a direct communication link connecting it to four other processors. Figure 2.16(a) illustrates a two-dimensional mesh of nine processors. If both dimensions of the mesh contain an equal number of processors, then it is called a ***square mesh***; otherwise it is called a ***rectangular mesh***.

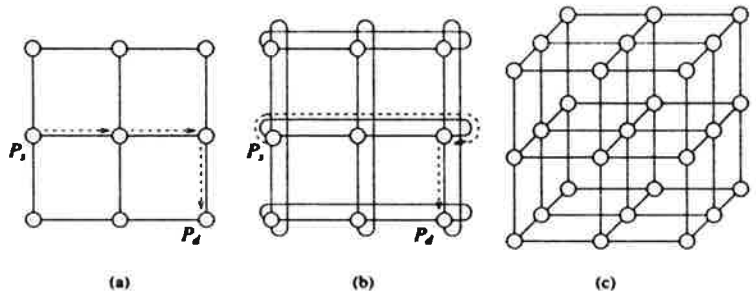

Figure 2.16 (a) A two-dimensional mesh with an illustration of routing a message from processor P_s to processor P_d; (b) a two-dimensional wraparound mesh with an illustration of routing a message from processor P_s to processor P_d; (c) a three-dimensional mesh.

Often, the processors at the periphery are connected by wraparound connections. Such a mesh is called a ***wraparound mesh*** or a ***torus*** (Figure 2.16(b)). A message from one processor to another can be routed in the mesh by first sending it along one dimension and then along the other dimension until it reaches its destination. Figures 2.16(a) and (b) illustrate this routing strategy for communicating a message between two processors. Common extensions of the two-dimensional mesh include the ***three-dimensional mesh*** and the ***three-dimensional wraparound mesh***. Figure 2.16(c) illustrates a three-dimensional mesh. Many commercially available parallel computers are based on the mesh network. These include two-dimensional meshes such as the DAP and Paragon XP/S, and three-dimensional meshes such as the Cray T3D and J-Machine. The Tera computer is based on a sparse three-dimensional mesh network (Figure 3.26).

Tree Network A ***tree network*** is one in which there is only one path between any pair of processors. Both linear arrays and star-connected networks are special cases of

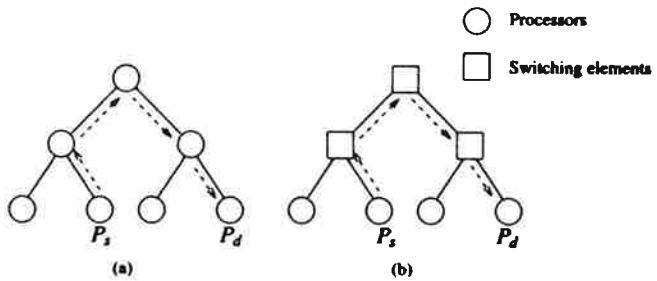

Figure 2.17 Complete binary tree networks and message routing in them.

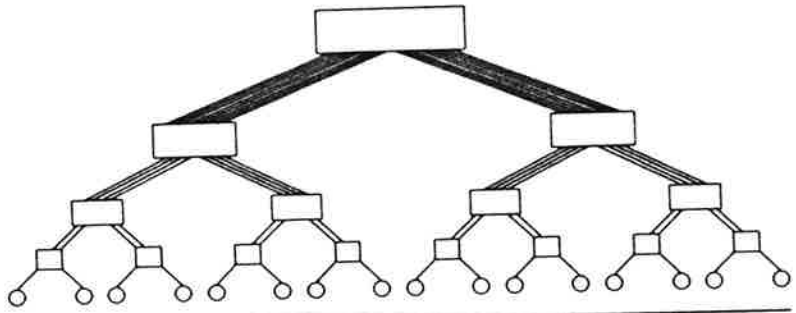

Figure 2.18 A fat tree network of 16 processors.

tree networks. Figure 2.17 shows networks based on complete binary trees. Static tree networks have a processor at each node of the tree (Figure 2.17(a)). Tree networks also have a dynamic counterpart. In a dynamic tree network, nodes at intermediate levels are switching elements and the leaf nodes are processors (Figure 2.17(b)).

To route a message in a tree, the source processor sends the message up the tree until it reaches the processor or the switch at the root of the smallest subtree containing both the source and destination processors. Then the message is sent down the tree toward the destination processor. This is illustrated in Figure 2.17.

Tree networks suffer from a communication bottleneck at higher levels of the tree. For example, when many processors in the left subtree of a node communicate with processors in the right subtree, the root node has to handle all the messages. This problem can be alleviated by increasing the number of communication links between processors that are closer to the root. This network, illustrated in Figure 2.18, is called a ***fat tree***.

The DADO parallel computer uses a static binary tree interconnection network. The CM-5 is based on a dynamic fat-tree network.

Hypercube Network A hypercube is a multidimensional mesh of processors with exactly two processors in each dimension. A d-dimensional hypercube consists of $p = 2^d$ processors. A hypercube can be recursively constructed as follows: a zero-dimensional hypercube is a single processor; a one-dimensional hypercube is constructed by connecting two zero-dimensional hypercubes; in general, a $(d + 1)$-dimensional hypercube is constructed by connecting the corresponding processors of two d-dimensional hypercubes.

Figure 2.19 shows hypercubes of dimensions zero to four. This figure also illustrates how we can use the recursive definition of a hypercube to label processors. When a $(d+1)$-dimensional hypercube is constructed by connecting two d-dimensional hypercubes, the labels of the processors of one hypercube are prefixed with a zero and those of the second hypercube are prefixed with a one.

Some of the important properties of a hypercube network are as follows:

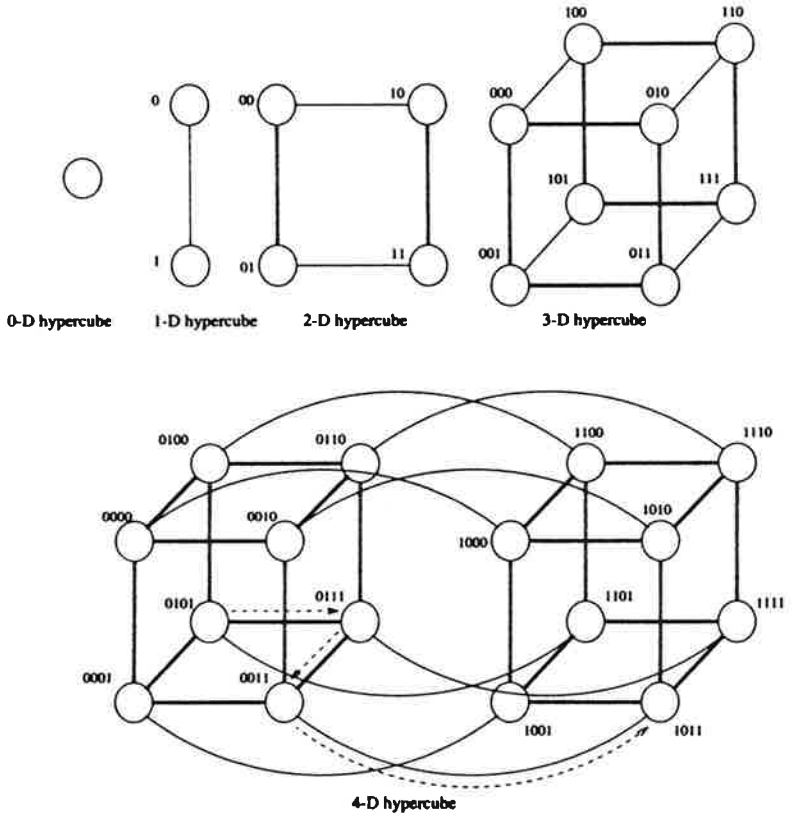

Figure 2.19 Hypercube-connected architectures of zero, one, two, three, and four dimensions. The figure also illustrates routing of a message from processor 0101 to processor 1011 in a four-dimensional hypercube.

(1) Two processors are connected by a direct link if and only if the binary representation of their labels differ at exactly one bit position. This is illustrated in Figure 2.19.

(2) In a d-dimensional hypercube, each processor is directly connected to d other processors.

(3) A d-dimensional hypercube can be partitioned into two $(d-1)$-dimensional subcubes as follows. Select a bit position and group together all the processors whose labels have a zero at the selected position; all of these processors make up one partition, and the remaining processors comprise the second partition. Since processor labels have d bits, d such partitions exist. This property follows directly from the recursive definition of the hypercube. Figure 2.20 illustrates the three

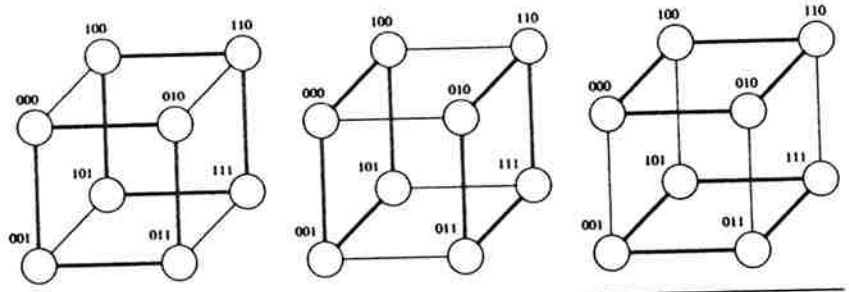

Figure 2.20 Three distinct partitions of a three-dimensional hypercube into two two-dimensional cubes. Links connecting processors within a partition are indicated by bold lines.

partitions of a three-dimensional hypercube. The bold lines in the figure connect the processors that belong to a partition.

(4) The processor labels in a d-dimensional hypercube contain d bits. Fixing any k of these bits, the processors that differ at the remaining $d-k$ bit positions form a $(d-k)$-dimensional subcube composed of $2^{(d-k)}$ processors (Problem 2.7). Since k bits can be fixed in 2^k different ways, there are 2^k such subcubes. Figure 2.21 illustrates this property. In this figure, $k=2$ and $d=4$. The four subcubes (of four processors each) that are formed by fixing the two most significant label bits are shown in Figure 2.21(a). The subcubes formed by fixing the two least significant bits are shown in Figure 2.21(b).

(5) Consider the labels s and t of two processors in a hypercube. The total number of bit positions at which these two labels differ is called the ***Hamming distance*** between them. For example, the Hamming distance between processors labeled 011 and 101 in a three-dimensional hypercube is two. Similarly, the Hamming distance between labels 101 and 010 is three. The Hamming distance between s and t is the number of bits that are one in the binary representation of $s \oplus t$, where $\oplus$ is the bitwise exclusive-or operation.
The number of communication links in the shortest path between two processors is the Hamming distance between their labels. A message can be routed from processor s to processor t by passing the message along dimensions that correspond to bit positions having a one in the binary representation of $s \oplus t$. Figure 2.19 illustrates routing of a message from processor s (labeled 0101) to processor t (labeled 1011). For this example, $s \oplus t$ is 1110. The message is routed along dimensions corresponding to bit positions one, two, and three (assuming the least significant bit is bit position zero). Since the binary representation of $s \oplus t$ can contain at most d ones, the shortest path between any two processors in a hypercube cannot have more than d links.

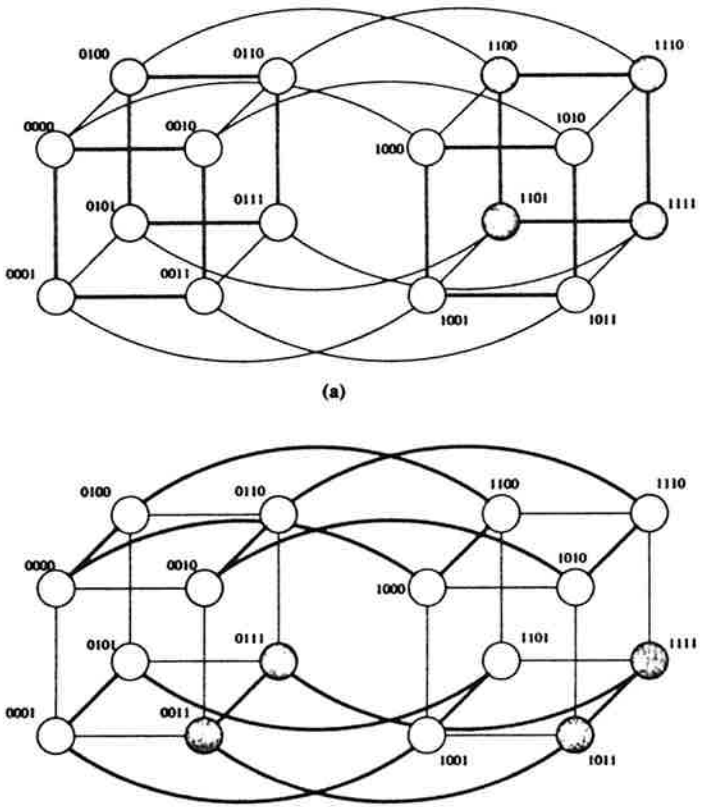

Figure 2.21 The two-dimensional subcubes of a four-dimensional hypercube formed by fixing the two most significant label bits (a) and the two least significant bits (b). Processors within a subcube are connected by bold lines.

Parallel computers based on the hypercube network include the nCUBE 2, Cosmic Cube, and iPSC.

k-ary d-cube Networks A d-dimensional hypercube, also called a binary d-cube, is a d-dimensional mesh with two processors along each dimension. A ring, in contrast, is a one-dimensional structure with p processors along its only dimension. These topologies define the extremes of a class of topologies called ***k-ary d-cubes***. Here, d is the ***dimension*** of the network and k is the ***radix***, which is defined as the number of processors along each dimension. The number of processors in the network, p, is equal to k^d. A ring of

p processors is a p-ary 1-cube. A two-dimensional wraparound mesh of p processors is a $\sqrt{p}$-ary 2-cube. A k-ary d-cube can be constructed from k k-ary $(d-1)$-cubes by connecting the processors that occupy identical positions in the cubes into rings.

2.4.2 Evaluating Static Interconnection Networks

This section discusses the criteria that characterize the cost and performance of static interconnection networks. We use these criteria to evaluate the networks introduced in the previous subsection.

Diameter The ***diameter*** of a network is the maximum distance between any two processors in the network. The distance between two processors is defined as the shortest path (in terms of number of links) between them. Since distance largely determines communication time, networks with smaller diameters are better. The diameter of a completely-connected network is one, and that of a star-connected network is two. The diameter of a ring network is $\lfloor p/2 \rfloor$. The diameter of a two-dimensional mesh without wraparound connections is $2(\sqrt{p}-1)$ and that of a wraparound mesh is $2\lfloor \sqrt{p}/2 \rfloor$. The diameter of a hypercube-connected network is $\log p$. The diameter of a complete binary tree is $2\log((p+1)/2)$ because the two communicating processors may be in separate subtrees of the root node, and a message might have to travel all the way to the root and then down the other subtree.

Connectivity The ***connectivity*** of a network is a measure of the multiplicity of paths between any two processors. A network with high connectivity is desirable, because it lowers contention for communication resources. One measure of connectivity is the minimum number of arcs that must be removed from the network to break it into two disconnected networks. This is called the ***arc connectivity*** of the network. The arc connectivity is one for linear arrays, as well as tree and star networks. It is two for rings and 2-D meshes without wraparound, four for 2-D wraparound meshes, and d for d-dimensional hypercubes.

Bisection Width and Bisection Bandwidth The ***bisection width*** of a network is defined as the minimum number of communication links that have to be removed to partition the network into two equal halves. The bisection width of a ring is two, since any partition cuts across only two communication links. Similarly, the bisection width of a two-dimensional p-processor mesh without wraparound connections is $\sqrt{p}$ and with wraparound connections is $2\sqrt{p}$. The bisection width of a tree and a star is one, and that of a completely-connected network of p processors is $p^2/4$. The bisection width of a hypercube can be derived from its construction. We construct a d-dimensional hypercube by connecting corresponding links of two $(d-1)$-dimensional hypercubes. Since each of these subcubes contains $2^{(d-1)}$ or $p/2$ processors, at least $p/2$ communication links must cross any partition of a hypercube into two subcubes (Problem 2.9).

The number of bits that can be communicated simultaneously over a link connecting two processors is called the ***channel width***. Channel width is equal to the number of

Table 2.1 A summary of the characteristics of various static network topologies connecting p processors.

Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of links)
Completely-connected	1	$p^2/4$	$p-1$	$p(p-1)/2$
Star	2	1	1	$p-1$
Complete binary tree	$2\log((p+1)/2)$	1	1	$p-1$
Linear array	$p-1$	1	1	$p-1$
Ring	$\lfloor p/2 \rfloor$	2	2	p
2-D mesh without wraparound	$2(\sqrt{p}-1)$	$\sqrt{p}$	2	$2(p-\sqrt{p})$
2-D wraparound mesh	$2\lfloor \sqrt{p}/2 \rfloor$	$2\sqrt{p}$	4	$2p$
Hypercube	$\log p$	$p/2$	$\log p$	$(p \log p)/2$
Wraparound k-ary d-cube	$d\lfloor k/2 \rfloor$	$2k^{d-1}$	$2d$	dp

physical wires in each communication link. The peak rate at which a single physical wire can deliver bits is called the ***channel rate***. The peak rate at which data can be communicated between the ends of a communication link is called ***channel bandwidth***. Channel bandwidth is the product of channel rate and channel width.

The ***bisection bandwidth*** of a network is defined as the minimum volume of communication allowed between any two halves of the network with an equal number of processors. It is the product of the bisection width and the channel bandwidth.

Cost Many criteria can be used to evaluate the cost of a network. One way of defining the cost of a network is in terms of the number of communication links or the number of wires required by the network. Linear arrays and trees use only $p-1$ links to connect p processors. A d-dimensional wraparound mesh has dp links. A hypercube-connected network has $(p \log p)/2$ links.

The bisection bandwidth of a network can also be used as a measure of its cost, as it provides a lower bound on the area in a two-dimensional packaging or the volume in a three-dimensional packaging. If the bisection width of a network is w, the lower bound on the area in a two-dimensional packaging is $\Theta(w^2)$, and the lower bound on the volume in a three-dimensional packaging is $\Theta(w^{3/2})$. According to this criterion, hypercubes and completely connected networks are more expensive than the other networks.

We summarize the characteristics of various networks in Table 2.1. There is no single network that is superior on the basis of all the criteria. We must make tradeoffs and select a network on the basis of both the system's cost and its intended applications.

2.5 Embedding Other Networks into a Hypercube

Given two graphs, $G(V, E)$ and $G'(V', E')$, embedding graph G into graph G' maps each vertex in the set V onto a vertex (or a set of vertices) in set V' and each edge in the set E onto an edge (or a set of edges) in E'. Let the nodes in a graph correspond to processors and the edges to communication links in an interconnection network. Embedding one graph into another is important because an algorithm may have been designed for a specific interconnection network, and it may be necessary to adapt it to another network.

When mapping graph $G(V, E)$ into $G'(V', E')$, three parameters are important. First, it is possible that more than one edge in E is mapped onto a single edge in E'. This leads to additional traffic on the corresponding communication link. The maximum number of edges mapped onto any edge in E' is called the ***congestion*** of the mapping. Second, an edge in E may be mapped onto multiple contiguous edges in E'. This is significant because traffic on the corresponding communication link must traverse more than one link and will incur longer delays. The maximum number of links in E' that any edge in E is mapped onto is called the ***dilation*** of the mapping. Third, the sets V and V' may contain different numbers of vertices. If so, a processor in V corresponds to more than one processor in V'. The ratio of the number of processors in the set V' to that in set V is called the ***expansion*** of the mapping.

In this section, we discuss embeddings of various interconnection networks into the hypercube. We limit the scope of the discussion to cases in which sets V and V' contain an equal number of processors (an expansion of one). Furthermore, all mappings presented in this section have at most one edge in E that maps onto one edge in E' (a congestion of one).

2.5.1 Embedding a Linear Array into a Hypercube

A linear array (or a ring) composed of 2^d processors (labeled 0 through $2^d - 1$) can be embedded into a d-dimensional hypercube by mapping processor i of the linear array onto processor $G(i, d)$ of the hypercube. The function $G(i, x)$ is defined as follows:

$$
\begin{aligned}
G(0,1) &= 0 \\
G(1,1) &= 1 \\
G(i, x+1) &= \begin{cases} G(i,x), & i < 2^x \\ 2^x + G(2^{x+1} - 1 - i, x), & i \geq 2^x \end{cases}
\end{aligned}
$$

The function G is called the ***binary reflected Gray code*** (RGC). The entry $G(i, d)$ denotes the i^{th} entry in the sequence of Gray codes of d bits. Gray codes of $d+1$ bits are derived from a table of Gray codes of d bits by reflecting the table and prefixing the reflected entries with a one and the original entries with a zero. This process is illustrated in Figure 2.22(a).

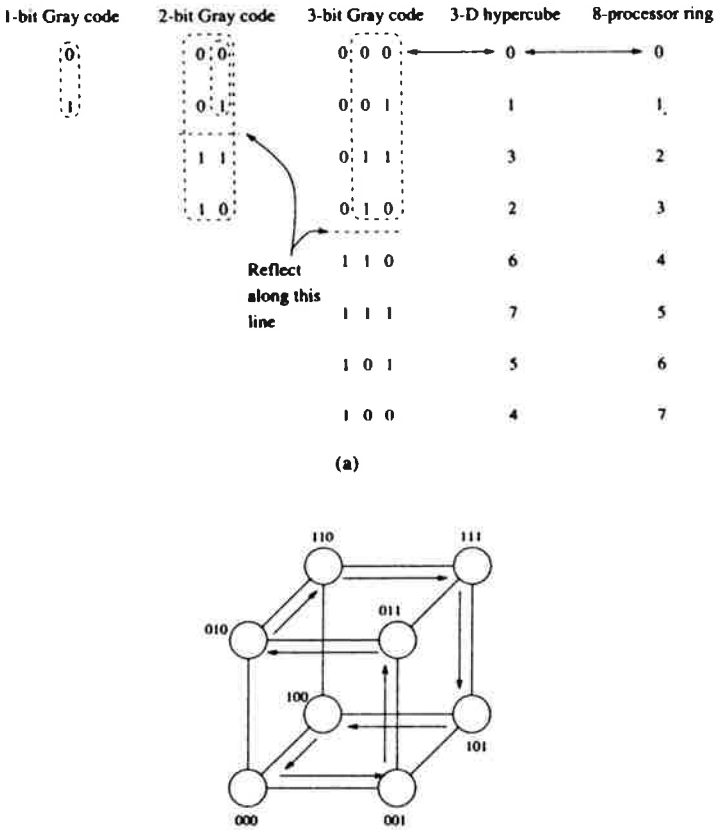

Figure 2.22 A three-bit reflected Gray code ring (a) and its embedding into a three-dimensional hypercube (b).

A careful look at the Gray code table reveals that two adjoining entries ($G(i, d)$ and $G(i + 1, d)$) differ from each other at only one bit position. Since processor i in the linear array is mapped to processor $G(i, d)$, and processor $i + 1$ is mapped to $G(i + 1, d)$, there is a direct link in the hypercube that corresponds to each direct link in the linear array. (Recall that two processors whose labels differ at only one bit position have a direct link in a hypercube.) Therefore, the mapping specified by the function G has a dilation of one and a congestion of one. Figure 2.22(b) illustrates the embedding of an eight-processor ring into a three-dimensional hypercube.

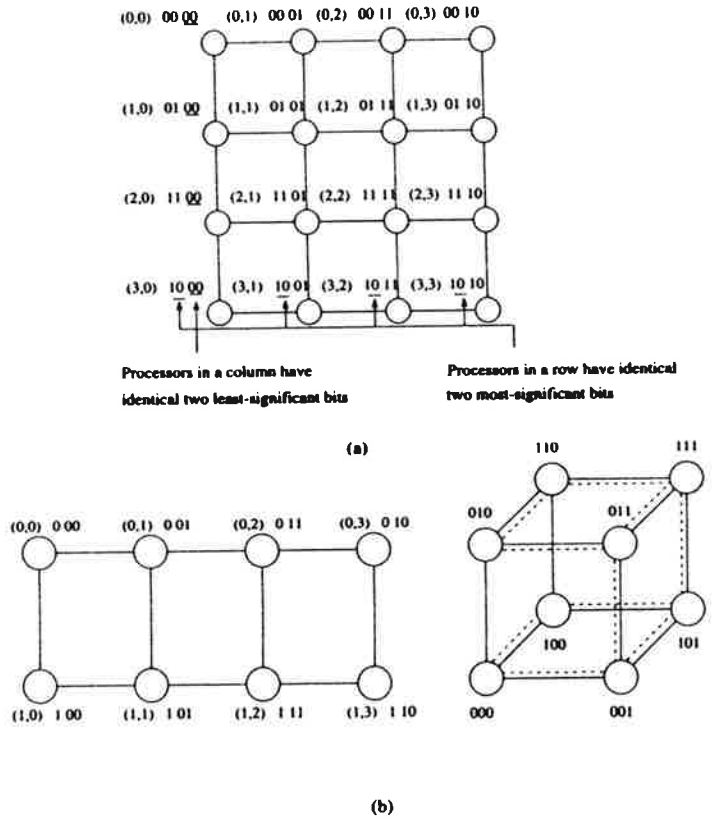

Figure 2.23 (a) A 4 × 4 mesh illustrating the mapping of mesh processors to processors in a four-dimensional hypercube; and (b) a 2 × 4 processor mesh embedded into a three-dimensional hypercube.

2.5.2 Embedding a Mesh into a Hypercube

Embedding a mesh into a hypercube is a natural extension of embedding a ring into a hypercube. We can embed a $2^r \times 2^s$ wraparound mesh into a 2^{r+s}-processor hypercube by mapping processor (i, j) of the mesh onto processor $G(i, r) \| G(j, s)$ of the hypercube (where $\|$ denotes concatenation of the two Gray codes). Note that immediate neighbors in the mesh are mapped to hypercube processors whose processor labels differ in exactly one bit position. Therefore, this mapping has a dilation of one and a congestion of one.

For example, consider embedding a 2 × 4 mesh into an eight-processor hypercube. The values of r and s are one and two, respectively. Processor (i, j) of the mesh is mapped to processor $G(i, 1) \| G(j, 2)$ of the hypercube. Therefore, processor (0, 0) of the

mesh is mapped to processor 000 of the hypercube, because $G(0, 1)$ is 0 and $G(0, 2)$ is 00; concatenating the two yields the label 000 for the hypercube processor. Similarly, processor (0, 1) of the mesh is mapped to processor 001 of the hypercube, and so on. Figure 2.23 illustrates embedding meshes into hypercubes.

This mapping of a mesh into a hypercube has certain useful properties. All processors in the same row of the mesh are mapped to hypercube processors whose labels have r identical most significant bits. We know from Section 2.4.1 that fixing any r bits in the processor label of an $(r + s)$-dimensional hypercube yields a subcube of dimension s with 2^s processors. Since each mesh processor is mapped onto a unique processor in the hypercube, and each row in the mesh has 2^s processors, every row in the mesh is mapped to a distinct subcube in the hypercube. Similarly, each column in the mesh is mapped to a distinct subcube in the hypercube.

2.5.3 Embedding a Binary Tree into a Hypercube

Binary trees can be embedded into hypercubes in several ways. Consider a complete binary tree of depth d containing processors only at its leaf nodes. One natural embedding of this tree into a 2^d-processor hypercube is as follows:

(1) The root of the tree is mapped onto any hypercube processor.
(2) For each processor i at depth j, the left child of i is mapped to hypercube processor i, and the right child of i is mapped to the hypercube processor whose label we obtain by inverting bit j of processor i's label (that is, to processor $i \oplus 2^{(j-1)}$, where $\oplus$ is the bitwise logical exclusive-or operation).

Since we determine the right child of a node by inverting only one bit, there is an edge in the hypercube that connects these two processors. Figure 2.24 illustrates an eight-processor tree embedded into a three-dimensional hypercube. In the mapping shown in the figure, the root of the tree is mapped onto processor 011 of the hypercube.

If processors are assumed to be at the leaves of the tree, then each edge between a pair of distinct nodes is mapped onto a single edge in the hypercube. Furthermore, since the number of processors in the hypercube and the tree are equal, the expansion of this mapping is one. Therefore, this mapping has a congestion, dilation, and expansion of one. If, however, the internal nodes of the tree are also processors, then this mapping cannot be used (Problem 2.17).

2.6 Routing Mechanisms for Static Networks

Efficient algorithms for routing a message to its destination are critical to the performance of parallel computers. A *routing mechanism* determines the path a message takes through the network to get from the source to the destination processor. It takes as input a message's

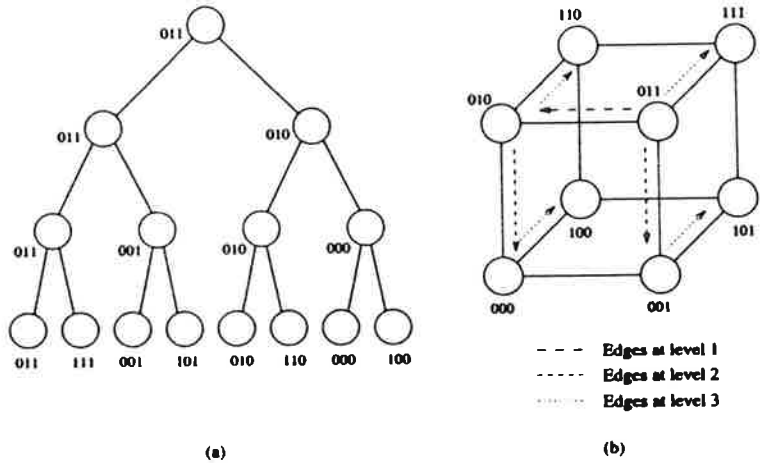

Figure 2.24 A tree rooted at processor 011 (=3) and embedded into a three-dimensional hypercube: (a) the organization of the tree rooted at processor 011, and (b) the tree embedded into a three-dimensional hypercube.

source and destination processors. It may also use information about the state of the network. It returns one or more paths through the network from the source to the destination processor.

Routing mechanisms can be classified as ***minimal*** or ***nonminimal***. A minimal routing mechanism always selects one of the shortest paths between the source and the destination. In a minimal routing scheme, each link brings a message closer to its destination, but the scheme can lead to congestion in parts of the network. A nonminimal routing scheme, in contrast, may route the message along a longer path to avoid network congestion.

Routing mechanisms can also be classified on the basis of how they use information regarding the state of the network. A ***deterministic routing*** scheme determines a unique path for a message, based on its source and destination. It does not use any information regarding the state of the network. Deterministic schemes may result in uneven use of the communication resources in a network. In contrast, an ***adaptive routing*** scheme uses information regarding the current state of the network to determine the path of the message. Adaptive routing detects congestion in the network and routes messages around it.

One commonly used deterministic minimal routing technique is called ***dimension-ordered routing***. Dimension-ordered routing assigns successive channels for traversal by a message based on a numbering scheme determined by the dimension of the channel. The dimension-ordered routing technique for a two-dimensional mesh is called ***XY-routing*** and that for a hypercube is called ***E-cube routing***.

Consider a two-dimensional mesh without wraparound connections. In the XY-routing scheme, a message is sent first along the X dimension until it reaches the column

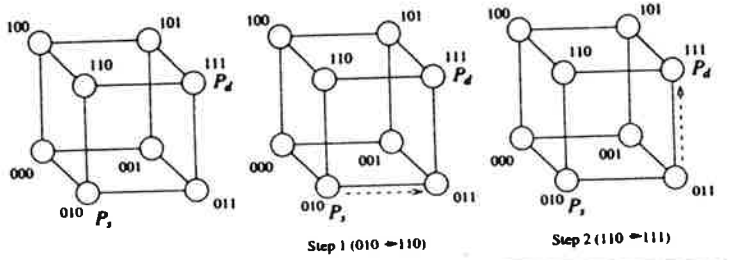

Figure 2.25 Routing a message from processor P_s (010) to processor P_d (111) in a three-dimensional hypercube using E-cube routing.

of the destination processor and then along the Y dimension until it reaches its destination. Let $P_{Sy,Sx}$ represent the position of the source processor and $P_{Dy,Dx}$ represent the position of the destination processor. Any minimal routing scheme should return a path of length $|Sx-Dx|+|Sy-Dy|$. Assume that $Dx \geq Sx$ and $Dy \geq Sy$. In the XY-routing scheme, the message is passed through intermediate processors $P_{Sy,Sx+1}, P_{Sy,Sx+2}, \ldots, P_{Sy,Dx}$ along the X dimension and then through processors $P_{Sy+1,Dx}, P_{Sy+2,Dx}, \ldots, P_{Dy,Dx}$ along the Y dimension to reach the destination. Note that the length of this path is indeed $|Sx - Dx| + |Sy - Dy|$. The routing scheme for meshes discussed in Section 2.4.1 is an example of XY-routing.

E-cube routing for hypercube-connected networks works similarly. Consider a d-dimensional hypercube of p processors. Let P_s and P_d be the labels of the source and destination processors. We know from Section 2.4.1 that the binary representations of these labels are d bits long. Furthermore, the minimum distance between these processors is given by the number of ones in $P_s \oplus P_d$ (where $\oplus$ represents the bitwise exclusive-OR operation). In the E-cube algorithm, processor P_s computes $P_s \oplus P_d$ and sends the message along dimension k, where k is the position of the least significant nonzero bit in $P_s \oplus P_d$. At each intermediate step, processor P_i, which receives the message, computes $P_i \oplus P_d$ and forwards the message along the dimension corresponding to the least significant nonzero bit. This process continues until the message reaches its destination. Example 2.1 illustrates E-cube routing in a three-dimensional hypercube network.

Example 2.1 E-cube Routing in a Hypercube Network

Consider the three-dimensional hypercube shown in Figure 2.25. Let $P_s = 010$ and $P_d = 111$ represent the source and destination processors for a message. Processor P_s computes $010 \oplus 111 = 101$. In the first step, P_s forwards the message along the dimension corresponding to the least significant bit to processor 011. Processor 011 sends the message along the dimension corresponding to the most significant bit ($011 \oplus 111 = 100$). The message reaches processor 111, which is the destination of the message. ■

In the rest of this book we assume deterministic and minimal message routing for analyzing parallel algorithms. Furthermore, unless otherwise mentioned, routing in hypercubes is assumed to be based on the E-cube algorithm and that in meshes on the XY-routing scheme.

2.7 Communication Costs in Static Interconnection Networks

The time spent communicating information from one processor to another is often a major source of overhead when executing programs on a parallel computer. The time taken to communicate a message between two processors in the network is called ***communication latency***. Communication latency is the sum of the time to prepare a message for transmission and the time taken by the message to traverse the network to its destination. The principal parameters that determine the communication latency are as follows:

(1) ***Startup time*** (t_s): The startup time is the time required to handle a message at the sending processor. This includes the time to prepare the message (adding header, trailer, and error correction information), the time to execute the routing algorithm, and the time to establish an interface between the local processor and the router. This delay is incurred only once for a single message transfer.

(2) ***Per-hop time*** (t_h): After a message leaves a processor, it takes a finite amount of time to reach the next processor in its path. The time taken by the header of a message to travel between two directly-connected processors in the network is called the per-hop time. It is also known as ***node latency***. The per-hop time is directly related to the latency within the routing switch for determining which output buffer or channel the message should be forwarded to.

(3) ***Per-word transfer time*** (t_w): If the channel bandwidth is r words per second, then each word takes time $t_w = 1/r$ to traverse the link. This time is called the per-word transfer time.

Many factors influence the communication latency of a network, such as the topology of the network and the ***switching techniques***. We now discuss two switching techniques that are frequently used in parallel computers. The routing techniques that use them are called store-and-forward routing and cut-through routing.

2.7.1 Store-and-Forward Routing

In store-and-forward routing, when a message is traversing a path with multiple links, each intermediate processor on the path forwards the message to the next processor after it has received and stored the entire message. Figure 2.26(a) shows the communication of a message through a store-and-forward network.

Suppose that a message of size m is being transmitted through such a network. Assume that it traverses l links. At each link, the message incurs a cost t_h for the header

and mt_w for the rest of the message to traverse the link. Since there are l such links, the total time is $(t_h + mt_w)l$. Therefore, for store-and-forward routing, the total communication cost for a message of size m words to traverse l communication links is

$$t_{comm} = t_s + (mt_w + t_h)l. \tag{2.2}$$

In current parallel computers, the per-hop time t_h is quite small. For most parallel algorithms, it is less than mt_w even for small values of m and thus can be ignored. For parallel algorithms using store-and-forward routing discussed in this book, we simplify the time given by Equation 2.2 to

$$t_{comm} = t_s + mt_w l.$$

2.7.2 Cut-Through Routing

Store-and-forward routing makes poor use of communication resources. A message is sent from one processor to the next only after the entire message has been received (Figure 2.26(a)). Consider the scenario shown in Figure 2.26(b), in which the original message is broken into two equally sized parts before it is sent. In this case, an intermediate processor waits for only half of the original message to arrive before passing it on. The increased utilization of communication resources and reduced communication time is apparent from Figure 2.26(b). Figure 2.26(c) goes a step further and breaks the message into four parts.

This example demonstrates the underlying principle of ***cut-through routing***, in which a message is advanced from the incoming link to the outgoing link as it arrives. ***Wormhole routing*** is a type of cut-through routing. A message being communicated travels in small units called ***flow-control digits*** or ***flits***. In wormhole routing, flits are pipelined through the communication network. An intermediate processor does not wait for the entire message to arrive before forwarding it. As soon as a flit is received at an intermediate processor, the flit is passed on to the next processor. All flits in a message are routed along the same path. It is no longer necessary to have buffer space at each intermediate processor to store the entire message. Therefore, wormhole routing uses less memory and memory bandwidth at intermediate processors and is faster.

While traversing the network, if a message needs to use a link that is currently in use, then the message is blocked. This may lead to deadlock. Figure 2.27 illustrates deadlock in a wormhole routing network. The destinations of messages 0, 1, 2, and 3 are A, B, C, and D, respectively. A flit from message 0 occupies the link CB (and the associated buffers). However, since link BA is occupied by a flit from message 3, the flit from message 0 is blocked. Similarly, the flit from message 3 is blocked since link AD is in use. We can see that no messages can progress in the network and the network is deadlocked. Deadlocks can be avoided in wormhole networks by using appropriate routing techniques. Both E-cube and XY-routing algorithms are deadlock-free routing schemes.

Consider a message that is traversing such a network. If the message traverses l links, and t_h is the per-hop time, then the header of the message takes lt_h time to reach the

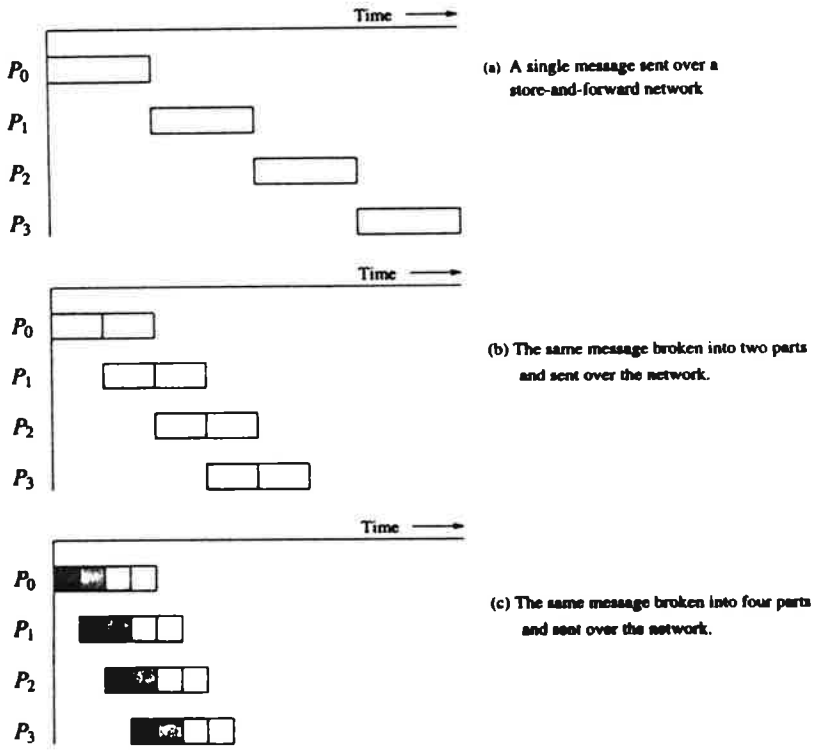

Figure 2.26 Passing a message from processor P_0 to P_3 (a) through a store-and-forward communication network; (b) and (c) extending the concept to cut-through routing. The shaded regions represent the time that the message is in transit. The startup time associated with this message transfer is assumed to be zero.

destination. If the message is m words long, then the entire message will arrive in time mt_w after the arrival of the header of the message. Therefore, the total communication time for cut-through routing is given by

$$t_{comm} = t_s + lt_h + mt_w.$$

This time is an improvement over store-and-forward routing. Disregarding the startup time, the cost to send a message of size m over l hops is only $\Theta(m + l)$ for cut-through routing, whereas in store-and-forward routing, it is $\Theta(ml)$. Note that if the communication is between nearest neighbors (that is, $l = 1$), or if the message size is small, then the communication time is similar for both routing schemes.

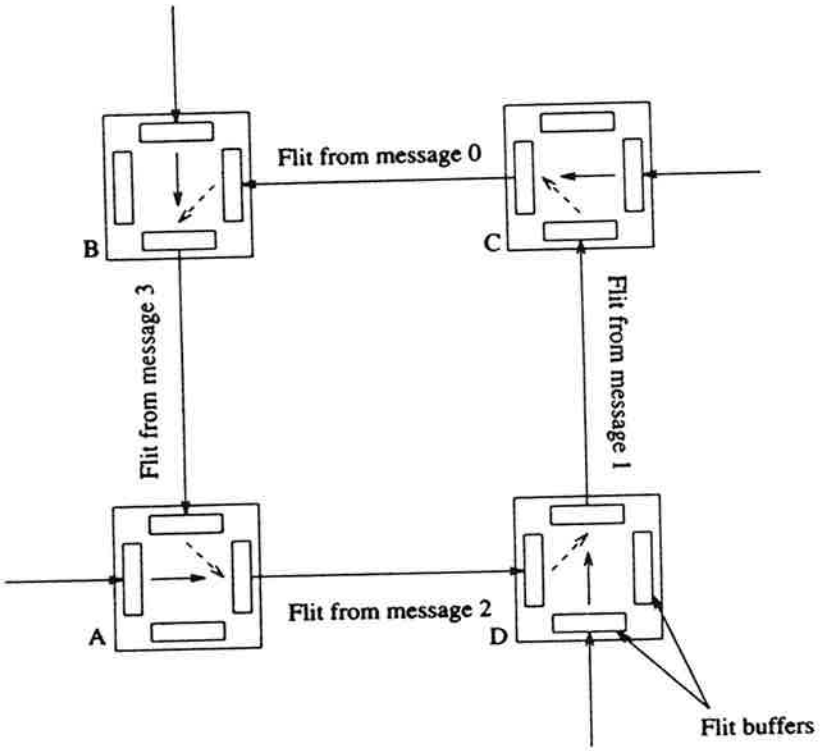

Figure 2.27 An example of deadlock in a wormhole-routing network.

2.8 Cost-Performance Tradeoffs

In this section, we show how various cost metrics can be used to investigate cost-performance tradeoffs in static networks. We illustrate this by analyzing the performance of a mesh and a hypercube network with identical costs.

If the cost of a network is proportional to the number of wires, then a square p-processor wraparound mesh with $(\log p)/4$ wires per channel costs as much as a p-processor hypercube with one wire per channel. Let us compare the average communication latencies of these two networks. The average distance l_{av} between any two processors in a two-dimensional wraparound mesh is $\sqrt{p}/2$ and that in a hypercube is $(\log p)/2$. The time for sending a message of size m between processors that are l_{av} hops apart is given by

$t_s + t_h l_{av} + t_w m$ in networks that use cut-through routing. Since the channel width of the mesh is scaled up by a factor of $(\log p)/4$, the per-word transfer time is reduced by the same factor. Hence, if the per-word transfer time on the hypercube is t_w, then the same time on a mesh with fattened channels is given by $4t_w/(\log p)$. Hence, the average communication latency for a hypercube is given by $t_s + t_h(\log p)/2 + t_w m$ and that for a wraparound mesh of the same cost is $t_s + t_h\sqrt{p}/2 + 4mt_w/(\log p)$.

Let us now investigate the behavior of these expressions. For a fixed number of processors, as the message size is increased, the communication term due to t_w dominates. Comparing t_w for the two networks, we see that the time for a wraparound mesh $(4mt_w/(\log p))$ is less than the time for a hypercube $(t_w m)$ if p is greater than 16 and the message size m is sufficiently large. Under these circumstances, point-to-point communication of large messages between random pairs of processors takes less time on a wraparound mesh with cut-through routing than on a hypercube of the same cost. Furthermore, for algorithms in which communication is suited to a mesh, the extra bandwidth of each channel results in better performance (Problem 3.33). Note that, with store-and-forward routing, the mesh is no longer more cost-efficient than a hypercube. Similar cost-performance tradeoffs can be analyzed for the general case of k-ary d-cubes (Problems 2.19–2.22).

The communication latencies above are computed under light load conditions in the network. As the number of messages increases, there is contention on the network. Contention affects the mesh network more adversely than the hypercube network. Therefore, if the network is heavily loaded, the hypercube will outperform the mesh (Problem 3.33).

If the cost of a network is proportional to its bisection width, then a p-processor wraparound mesh with $\sqrt{p}/4$ wires per channel has a cost equal to a p-processor hypercube with one wire per channel. Let us perform an analysis similar to the one above to investigate cost-performance tradeoffs using this cost metric. Since the mesh channels are wider by a factor of $\sqrt{p}/4$, the per-word transfer time will be lower by an identical factor. Therefore, the communication latencies for the hypercube and the mesh networks of the same cost are given by $t_s + t_h(\log p)/2 + t_w m$ and $t_s + t_h\sqrt{p}/2 + 4mt_w/\sqrt{p}$, respectively. Once again, as the message size m becomes large for a given number of processors, the t_w term dominates. Comparing this term for the two networks, we see that for $p > 16$ and sufficiently large message sizes, a mesh outperforms a hypercube of the same cost. Therefore, for large enough messages, a mesh is always better than a hypercube of the same cost, provided the network is lightly loaded (Problem 3.33). Even when the network is heavily loaded, the performance of a mesh is similar to that of a hypercube of the same cost (Problem 3.33).

2.9 Architectural Models for Parallel Algorithm Design

Different parallel architectures may require different algorithms to solve the same problem efficiently. Is it possible to avoid designing a new algorithm for each architecture? This would be possible if there was a universal abstract model for parallel architectures. Parallel programs designed for this universal model could be ported to specific computers.

The von Neumann model is a universal computational model for sequential computers, enabling the design of portable sequential programs. Although the processor architecture and memory organization of sequential computers differ, they all conform to the same abstract model. Consequently, a program designed for the von Neumann model can be compiled and run relatively efficiently on any sequential computer. In particular, the asymptotic time complexity of an algorithm on different implementations of the von Neumann model is the same. Indeed, the existence of a universal model has played a key role in the impressive growth of the application of sequential computers. If algorithms had to be redesigned for every new sequential architecture, the cost of software development would have been much higher, greatly limiting the cost-effective application of computers.

Universal Models for Parallel Computers A universal model for parallel computers must exhibit two characteristics. First, it must be sufficiently general to capture the salient properties of a large class of parallel computers. Second, programs designed for this abstract model must execute efficiently on actual parallel computers. These characteristics represent conflicting goals. A universal model for parallel computers must make assumptions about the degree of connectivity among the processors. Networks such as linear arrays and meshes have a relatively low degree of connectivity compared to networks such as hypercubes. If the model assumes a low degree of connectivity, algorithms designed for it will be optimal for parallel computers with this characteristic. However, the same algorithms may be sub-optimal for architectures with a high degree of connectivity, because the communication resources of the computer will be under-utilized. On the other hand, if the model assumes a high degree of connectivity among processors, then algorithms designed for the model may be sub-optimal on an architecture such as a mesh, yet optimal on a hypercube. Thus, regardless of the connectivity of the parallel model, it yields sub-optimal performance for some architectures. This argues against the existence of a universal parallel model. No such model is currently known to exist.

Another approach to parallel algorithm design is to design parallel algorithms in terms of basic data communication operations. In this approach, only the implementation of these operations must be optimized for different parallel computers. In practice, a relatively small set of communication operations form the core of many parallel algorithms. Therefore, designing these operations is a relatively simple task. This is the approach to parallel program design that is adopted in the rest of this book.

Role of Data Locality There are two major components of parallel algorithm design. The first one is the identification and specification of the overall problem as a set of tasks that can be performed concurrently. The second is the mapping of these tasks onto different processors so that the overall communication overhead is minimized. The first component specifies ***concurrency***, and the second one specifies ***data locality***. The performance of an algorithm on a parallel architecture depends on both. Concurrency is necessary to keep the processors busy. Locality is important because it minimizes communication overhead. Ideally, a parallel algorithm should have maximum concurrency and locality. However, for most algorithms, there is a tradeoff. An algorithm that has more concurrency often has less locality. As illustrated in Problem 5.36, data locality is important not only in message-passing architectures but also in shared-address-space architectures.

Architectural Models Used in This Book In this book, we discuss parallel algorithms mainly in the context of mesh- and hypercube-connected parallel computers. Although extensive work has been done on PRAM algorithms, we use the PRAM model only to identify concurrency in a problem. The PRAM model captures concurrency in a task; however, it fails to enforce data locality in parallel algorithms. In particular, a PRAM algorithm tells us very little about mapping the computation onto actual processors. This is because communication is free on a PRAM, but is often the most significant overhead on practical parallel computers.

The selection of the mesh topology follows from the following characteristics:

(1) Meshes containing a large number of processors can be constructed relatively inexpensively.
(2) Many applications map naturally onto a mesh network.
(3) Although the mesh network has a high diameter, as message size becomes large, the effect of higher diameter is diminished for networks using cut-through routing. Consequently, many algorithms ported directly from networks with a higher degree of connectivity (such as hypercubes) yield similar performance on mesh architectures. For these algorithms, meshes have a favorable cost-performance ratio.
(4) Several commercially-available parallel computers are based on the mesh network.

The selection of the hypercube was motivated by the following criteria:

(1) For a large class of problems, the fastest hypercube algorithms are asymptotically as fast as the fastest PRAM algorithms. Therefore, hypercubes are able not only to tap maximum concurrency for these applications but also to impose data locality.
(2) For many problems, the best hypercube algorithm is also the best algorithm for other networks such as fat trees, meshes, and multistage networks, provided the algorithm is adapted to them appropriately. Therefore, a hypercube algorithm is often a good starting point for designing algorithms for such architectures.
(3) Hypercubes have an elegant recursive structure that makes them attractive for designing a wide class of algorithms.

2.10 Bibliographic Remarks

Several textbooks discuss parallel architectures and interconnection networks [AG89, DeC89, Hwa93, HB84, Lil92, Sie85, Sto93]. The classification of parallel computers as SISD, SIMD, and MIMD was introduced by Flynn [Fly72]. He also proposed the MISD (multiple instruction stream, single data stream) model. MISD is less natural than the

other classes, although it can be viewed as a model for pipelining. Gordon Bell [Bel92] classifies MIMD parallel computers based on the physical and logical memory organization. Ni [Ni91] provides a layered classification of parallel computers based on hardware architecture, address space, communication model, language, programming environment, and applications.

Interconnection networks have been an area of active interest for decades. Feng [Fen81] provides a tutorial on static and dynamic interconnection networks. The perfect shuffle interconnection pattern was introduced by Stone [Sto71]. Omega networks were introduced by Lawrie [Law75]. Other multistage networks have also been proposed. These include the Flip network [Bat76] and the Baseline network [WF80]. Mesh of trees and pyramidal mesh are discussed by Leighton [Lei92]. Leighton [Lei92] provides a detailed discussion of many such networks.

The C.mmp was an early research prototype MIMD shared-address-space parallel computer based on the Crossbar switch [WB72]. The Fujitsu VPP 500 and the Cray Y-MP are more recent examples of crossbar-based parallel computers. Several parallel computers were based on multistage interconnection networks including the BBN Butterfly [BBN89], the NYU Ultracomputer [GGK+83], and the IBM RP-3 [PBG+85]. The Stanford Dash [LLG+92] and the KSR-1 [Ken90] are NUMA shared-address-space computers.

The Cosmic Cube [Sei85] was among the first message-passing parallel computer based on a hypercube-connected network. Commercially available hypercube-based computers include the nCUBE 2 [nCU90], the Intel iPSC-1, iPSC-2, and iPSC/860. Saad and Shultz [SS88, SS89a] derive interesting properties of the hypercube-connected network and a variety of other static networks [SS89b]. Many message-passing computers are based on the mesh network. The Intel Paragon XP/S [Sup91] and the Mosaic C [Sei92] are two-dimensional mesh-based computers. The MIT J-Machine [D+92] is based on a three-dimensional mesh network. The performance of mesh-connected computers can be improved by augmenting the mesh network with broadcast buses [KR87]. The reconfigurable mesh architecture (Figure 2.29) was introduced by Miller et al. [MKRS88].

The DADO parallel computer was based on a tree network [SM86]. It used a complete binary tree of depth ten. Leiserson [Lei85] introduced the fat-tree interconnection network and proved several interesting characteristics of it. He showed that for a given volume of hardware, no network has much better performance than a fat tree. The Thinking Machines CM-5 [Thi91] parallel computer is based on a fat tree interconnection network.

The Illiac IV [Bar68] was among the first SIMD parallel computers. Other SIMD computers include the Goodyear MPP [Bat80], the DAP 610, and more recently the CM-2 [Thi90], MasPar MP-1, and MasPar MP-2 [Nic90]. The CM-5 and DADO incorporate both SIMD and MIMD features. Both are MIMD computers but have extra hardware for fast synchronization, which enables them to operate in SIMD mode. The CM-5 has a control network to augment the data network. The control network provides such functions as broadcasting, reduction, and other global operations.

Leighton [Lei92] and Ranka and Sahni [RS90] discuss embedding one interconnection network into another. Gray codes, used in embedding linear array and mesh topologies, are discussed by Reingold [RND77]. Ranka and Sahni [RS90] discuss the concepts of congestion, dilation, and expansion. Our discussion of embeddings in Section 2.5 is influenced by Ranka and Sahni's discussion [RS90].

A comprehensive survey of cut-through routing techniques is provided by Ni and McKinley [NM93]. The wormhole routing technique was proposed by Dally and Seitz [DS86]. A related technique called ***virtual cut-through***, in which communication buffers are provided at intermediate processors, was described by Kermani and Kleinrock [KK79]. Dally and Seitz [DS87] discuss deadlock-free wormhole routing based on channel dependence graphs. Deterministic routing schemes based on dimension ordering are often used to avoid deadlocks. Cut-through routing has been used in several parallel computers including the nCUBE 2, Paragon XP/S, and J-Machine. Our discussion of wormhole routing is based on the paper by by Ni and McKinley [NM93]. The E-cube routing scheme for hypercubes was proposed by [SB77].

Dally [Dal90b] discusses the cost-performance tradeoffs of networks for message-passing computers. By using the bisection bandwidth of a network as a measure of the cost of the network, he showed that low-dimensional networks (such as two-dimensional meshes) are more cost-effective than high-dimensional networks (such as hypercubes) [Dal87, Dal90b, Dal90a]. Kreeger and Vempaty [KV92] derive the bandwidth equalization factor for a mesh with respect to a hypercube-connected computer for all-to-all personalized communication (Section 3.5). Gupta and Kumar [GK93] analyze the cost-performance tradeoffs of FFT computations on mesh and hypercube networks for two different cost criteria.

The properties of PRAMs have been studied extensively [FW78, KR88, LY86, Sni82, Sni85]. Books by Akl [Akl89], Gibbons [GR90], and Jaja [Jaj92] address PRAM algorithms. Our discussion of PRAM is based upon the book by Jaja [Jaj92]. A number of processor networks have been proposed to simulate PRAM models [AHMP87, HP89, LPP88, LPP89, MV84, Upf84, UW84]. Mehlhorn and Vishkin [MV84] propose the ***module parallel computer*** (MPC) to simulate PRAM models. The MPC is a message-passing parallel computer composed of p processors, each with a fixed amount of memory and connected by a completely-connected network. The MPC is capable of probabilistically simulating T steps of a PRAM in $T \log p$ steps if the total memory is increased by a factor of $\log p$. The main drawback of the MPC model is that a completely-connected network is difficult to construct for a large number of processors. Alt et al. [AHMP87] propose another model called the ***bounded-degree network*** (BDN). In this network, each processor is connected to a fixed number of other processors. Karlin and Upfal [KU86] describe an $O(T \log p)$ time probabilistic simulation of a PRAM on a BDN. Hornick and Preparata [HP89] propose a bipartite network that connects sets of processors and memory pools. They investigate both the message-passing MPC and BDN based on a mesh of trees.

Many modifications of the PRAM model have been proposed that attempt to bring it closer to practical parallel computers. Aggarwal, Chandra, and Snir [ACS89b, ACS89c]

propose the LPRAM (local-memory PRAM) model and the BPRAM (block PRAM) model [ACS89b]. They also introduce a hierarchical memory model of computation [ACS89a]. In this model, memory units at different levels are accessed in different times. Parallel algorithms for this model induce locality by bringing data into faster memory units before using them and returning them to the slower memory units. Other PRAM models such as phase PRAM [Gib89], XPRAM [Val90b], and the delay model [PY88] have also been proposed. Many researchers have investigated abstract universal models for parallel computers [C+93, Sny86, Val90a].

Problems

2.1 As mentioned in Section 2.2, CRCW is the most powerful PRAM model. We also discussed four classes of concurrent write operations for CRCW PRAMs: common CRCW, arbitrary CRCW, priority CRCW, and sum CRCW. It is possible to emulate one CRCW PRAM model on the other PRAM models. Let t be the run time of an algorithm on a priority CRCW PRAM model. Give an upper bound on the run time of this algorithm on the following models:

(a) a common CRCW PRAM

(b) an arbitrary CRCW PRAM

(c) a CREW PRAM

(d) an EREW PRAM

2.2 Consider an EREW PRAM with p processors and m memory locations. We can emulate this model on a p-processor message-passing parallel computer in which each processor has m/p memory locations. Let t be the run time of an algorithm on a p-processor EREW PRAM model. Give an upper bound on the run time of this algorithm on the following architectures:

(a) a p-processor ring

(b) a p-processor mesh

(c) a p-processor hypercube

2.3 [Lei92] The ***butterfly network*** is an interconnection network composed of $\log p$ levels (as the omega network). In a butterfly network, each switching element i at a level l is connected to the identically numbered element at level $l+1$ and to a switching element whose number differs from itself only at the l^{th} most significant bit. Therefore, switching element S_i is connected to element S_j at level l if $j = i$ or $j = i \oplus (2^{\log p - l})$.
Figure 2.28 illustrates an eight-processor butterfly network. Show the equivalence of a butterfly network and an omega network.

Hint: Rearrange the switches of an omega network so that it looks like a butterfly network.

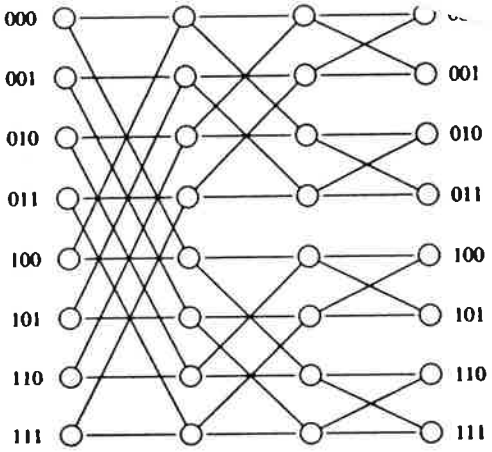

Figure 2.28 An eight-processor butterfly network.

2.4 Consider the omega network described in Section 2.3.3. As shown there, this network is a blocking network (that is, a processor that uses the network to access a memory location might prevent another processor from accessing another memory location). Consider an omega network that connects p processors. Define a function f that maps $P = [0, 1, \ldots, p-1]$ onto a permutation P' of P (that is, $P'[i] = f(P[i])$ and $P'[i] \in P$ for all $0 \leq i < p$). Think of this function as mapping communication requests by the processors so that processor $P[i]$ requests communication with processor $P'[i]$.

(a) How many distinct permutation functions exist?

(b) How many of these functions result in non-blocking communication?

(c) What is the probability that an arbitrary function will result in non-blocking communication?

2.5 Given the labeling scheme described in Section 2.4.1, how many distinct labelings exist for a d-dimensional hypercube?

2.6 A cycle in a graph is defined as a path originating and terminating at the same node. The length of a cycle is the number of edges in the cycle. Show that there are no odd-length cycles in a d-dimensional hypercube.

2.7 The labels in a d-dimensional hypercube use d bits. Fixing any k of these bits, show that processors whose labels differ in the remaining $d-k$ bit positions form a $(d-k)$-dimensional subcube composed of $2^{(d-k)}$ processors.

2.8 Let A and B be two processors in a d-dimensional hypercube. Define $H(A, B)$ to be the Hamming distance between A and B, and $P(A, B)$ to be the number of

distinct paths connecting A and B. These paths are called ***parallel paths*** and have no common processors other than A and B. Prove the following:

(a) The minimum distance in terms of communication links between A and B is given by $H(A, B)$.

(b) The total number of parallel paths between any two processors is

$$P(A, B) = d.$$

(c) The number of parallel paths between A and B of length $H(A, B)$ is

$$P_{length=H(A,B)}(A, B) = H(A, B).$$

(d) The length of the remaining $d - H(A, B)$ parallel paths is $H(A, B) + 2$.

2.9 In the informal derivation of the bisection width of a hypercube (Section 2.4.2), we used the construction of a hypercube to show that a d-dimensional hypercube is formed from two $(d-1)$-dimensional hypercubes. We argued that because corresponding processors in each of these subcubes have a direct communication link, there are $2^d - 1$ links across the partition. However, it is possible to partition a hypercube into two parts such that neither of the partitions is a hypercube. Show that any such partitions will have more than $2^d - 1$ direct links between them.

2.10 **[MKRS88]** A $\sqrt{p} \times \sqrt{p}$ ***reconfigurable mesh*** consists of a $\sqrt{p} \times \sqrt{p}$ array of processors connected to a grid-shaped reconfigurable broadcast bus. A 4×4 reconfigurable mesh is shown in Figure 2.29. Each processor has locally-controllable bus switches. The internal connections among the four ports, north (N), east (E), west (W), and south (S), of a processor can be configured during the execution of an algorithm. Note that there are 15 connection patterns. For example, {SW, EN}represents the configuration in which port S is connected to port W and port N is connected to port E. Each bit of the bus carries one of *1-signal* or *0-signal* at any time. The switches allow the broadcast bus to be divided into subbuses, providing smaller reconfigurable meshes. For a given set of switch settings, a ***subbus*** is a maximally-connected subset of the processors. Other than the buses and the switches, the reconfigurable mesh is similar to the standard two-dimensional mesh. Assume that only one processor is allowed to broadcast on a ***subbus*** shared by multiple processors at any time.
Determine the bisection width, the diameter, and the number of switching elements and communication links for a reconfigurable mesh of $\sqrt{p} \times \sqrt{p}$ processors. What are the advantages and disadvantages of a reconfigurable mesh as compared to a wraparound mesh?

2.11 **[Lei92]** A ***mesh of trees*** is a network that imposes a tree interconnection on a grid of processors. A $\sqrt{p} \times \sqrt{p}$ mesh of trees is constructed as follows. Starting with a $\sqrt{p} \times \sqrt{p}$ grid of processors a complete binary tree is imposed on each row of the grid. Then a complete binary tree is imposed on each column of the grid. Figure 2.30 illustrates the construction of a 4×4 mesh of trees. Assume that the nodes at intermediate levels are switching elements. Determine the bisection width, diameter, and total number of switching elements in a $\sqrt{p} \times \sqrt{p}$ mesh of processors.

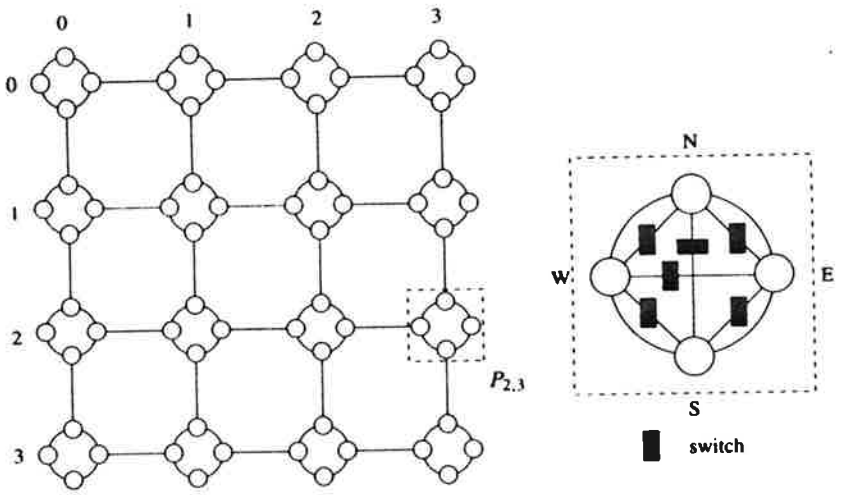

Figure 2.29 Switch connection patterns in reconfigurable mesh.

2.12 **[Lei92]** Extend the two-dimensional mesh of trees (Problem 2.11) to d dimensions to construct a $p^{1/d} \times p^{1/d} \times \cdots \times p^{1/d}$ mesh of trees. We can do this by fixing grid positions in all dimensions to different values and imposing a complete binary tree on the one dimension that is being varied.
Derive the total number of switching elements in a $p^{1/d} \times p^{1/d} \times \cdots \times p^{1/d}$ mesh of trees. Calculate the diameter, bisection width, and wiring cost in terms of the total number of wires. What are the advantages and disadvantages of a mesh of trees as compared to a wraparound mesh?

2.13 **[Lei92]** A network related to the mesh of trees is the d-dimensional ***pyramidal mesh***. A d-dimensional pyramidal mesh imposes a pyramid on the underlying processor grid (as opposed to a complete tree in the mesh of trees). The generalization is as follows. In the mesh of trees, all but one dimension are fixed and a tree is imposed on the remaining dimension. In a pyramid, all but two dimensions are fixed and a pyramid is imposed on the mesh formed by these two dimensions. In a tree, each node i at level k is connected to node $i/2$ at level $k-1$. Similarly, in a pyramid, a node (i, j) at level k is connected to a node $(i/2, j/2)$ at level $k-1$. Furthermore, the nodes at each level are connected in a mesh. A two-dimensional pyramidal mesh is illustrated in Figure 2.31.

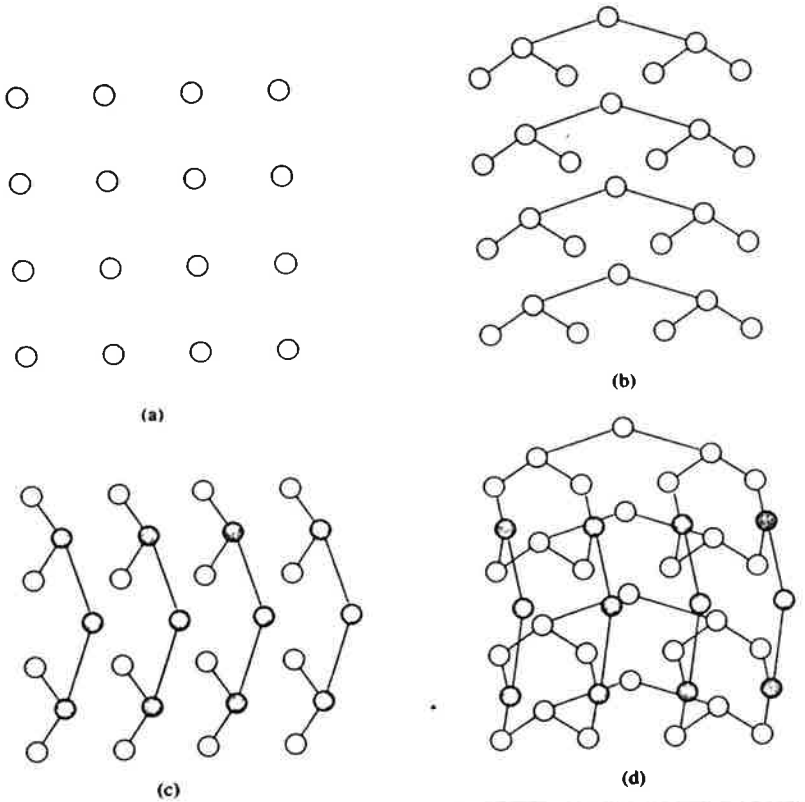

Figure 2.30 The construction of a 4 × 4 mesh of trees: (a) a 4 × 4 processor grid, (b) complete binary trees imposed over individual rows, (c) complete binary trees imposed over each column, and (c) the complete 4 × 4 mesh of trees.

For a $\sqrt{p} \times \sqrt{p}$ pyramidal mesh, assume that the intermediate nodes are switching elements, and derive the diameter, bisection width, arc connectivity, and cost in terms of the number of communication links and switching elements. What are the advantages and disadvantages of a pyramidal mesh as compared to a mesh of trees?

2.14 **[Lel92]** One of the drawbacks of a hypercube-connected network is that different wires in the network are of different lengths. This implies that data takes different times to traverse different communication links. It appears that two-dimensional mesh networks with wraparound connections suffer from this drawback too. However, it is possible to fabricate a two-dimensional wraparound mesh using wires of fixed length. Illustrate this layout by drawing such a 4 × 4 wraparound mesh.

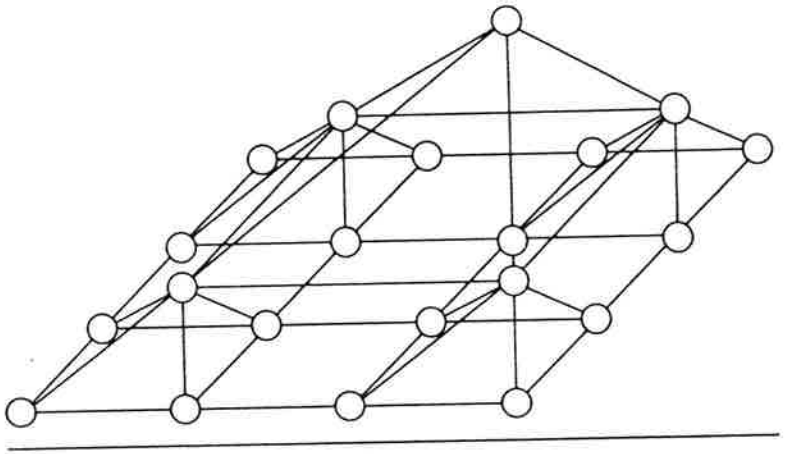

Figure 2.31 A 4 × 4 pyramidal mesh.

2.15 Show how to embed a p-processor three-dimensional mesh into a p-processor hypercube. What are the allowable values of p for your embedding?

2.16 Show how to embed a p-processor mesh of trees into a p-processor hypercube. Is your embedding different from the mesh embedding discussed in Section 2.5.2?

2.17 Consider a complete binary tree of $2^d - 1$ nodes in which each node is a processor. What is the minimum-dilation mapping of such a tree onto a d-dimensional hypercube?

2.18 The concept of a ***minimum congestion mapping*** is very useful. Consider two parallel computers with different interconnection networks such that a congestion-r mapping of the first into the second exists. Ignoring the dilation of the mapping, if each communication link in the second computer is more than r times faster than the first computer, the second computer is strictly superior to the first.
Now consider mapping a d-dimensional hypercube onto a 2^d-processor mesh. Ignoring the dilation of the mapping, what is the minimum-congestion mapping of the hypercube onto the mesh? Use this result to determine whether a 1024-processor mesh with communication links operating at 25 million bytes per second is strictly better than a 1024-processor hypercube (whose processors are identical to those used in the mesh) with communication links operating at two million bytes per second.

2.19 Derive the diameter, number of links, and bisection width of a k-ary d-cube with p processors. Define l_{av} to be the average distance between any two processors in the network. Derive l_{av} for a k-ary d-cube.

2.20 Consider a hypercube network of p processors. Assume that the channel width of each communication link is one. The channel width of the links in a k-ary d-cube

(for $d < \log p$) can be increased by equating the cost of the this network with that of a hypercube. Two distinct measures can be used to evaluate the cost of a network.

(1) The cost can expressed in terms of the total number of wires in the network (the total number of wires is a product of the number of communication links and the channel width).
(2) The bisection bandwidth can be used as a measure of cost.

Using each of these cost metrics and equating the cost of a k-ary d-cube with a hypercube, what is the channel width of a k-ary d-cube with an identical number of processors, channel rate, and cost?

2.21 The results from Problems 2.19 and 2.20 can be used in a cost-performance analysis of static interconnection networks. Consider a k-ary d-cube network of p processors with cut-through routing. Assume a hypercube-connected network of p processors with channel width one. The channel width of other networks in the family is scaled up so that their cost is identical to that of the hypercube. Let s and s' be the scaling factors for the channel width derived by equating the costs specified by the two cost metrics in Problem 2.20.
For each of the two scaling factors s and s', express the average communication time between any two processors as a function of the dimensionality (d) of a k-ary d-cube and the number of processors. Plot the communication time as a function of the dimensionality for $p = 256, 512$, and 1024, message size $m = 512$ bytes, $t_s = 50.0\mu$s, and $t_h = t_w = 0.5\mu$s (for the hypercube). For these values of p and m, what is the dimensionality of the network that yields the best performance for a given cost?

2.22 Repeat Problem 2.21 for a k-ary d-cube with store-and-forward routing.

References

[ACS89a] A. Aggarwal, A. K. Chandra, and M. Snir. A model for hierarchical memory. Technical Report RC 15118 (No. 67337), IBM T. J. Watson Research Center, Yorktown Heights, NY, 1989.

[ACS89b] A. Aggarwal, A. K. Chandra, and M. Snir. On communication latency in PRAM computations. Technical Report RC 14973 (No. 66882), IBM T. J. Watson Research Center, Yorktown Heights, NY, 1989.

[ACS89c] A. Aggarwal, A. K. Chandra, and M. Snir. Communication complexity of PRAMs. Technical Report RC 14998 (No. 64644), IBM T. J. Watson Research Center, Yorktown Heights, NY, Yorktown Heights, NY, 1989.

[AG89] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin/Cummings, Redwood City, CA, 1989.

[AHMP87] H. Alt, T. Hagerup, K. Mehlhorn, and F. P. Preparata. Deterministic simulation of idealized parallel computers on more realistic ones. *SIAM Journal of Computing*, 16(5):808–835, October 1987.

[Akl89] S. G. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice-Hall, Englewood Cliffs, NJ, 1989.

[Bar68] G. H. Barnes. The ILLIAC IV computer. *IEEE Transactions on Computers*, C-17(8):746–757, 1968.

[Bat76] K. E. Batcher. The Flip network in STARAN. In *Proceedings of International Conference on Parallel Processing*, 65–71, 1976.

[Bat80] K. E. Batcher. Design of a massively parallel processor. *IEEE Transactions on Computers*, 836–840, September 1980.

[BBN89] BBN Advanced Computers Inc. *TC-2000 Technical Product Summary*. Cambridge, MA. 1989.

[Bel92] C. G. Bell. Ultracomputer: A teraflop before its time. *Communications of the ACM*, 35(8):27–47, 1992.

[C+93] D. Culler et al. LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, 1–12, 1993. A detailed version is available as Technical Report UCB-CD-93-713, Computer Science Division, University of California, Berkeley, CA.

[D+92] W. J. Dally et al. The message-driven processor. *IEEE Micro*, 12(2):23–39, 1992.

[Dal87] W. J. Dally. *A VLSI Architecture for Concurrent Data Structures*. Kluwer Academic Publishers, Boston, MA, 1987.

[Dal90a] W. J. Dally. Analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 39(6), June 1990.

[Dal90b] W. J. Dally. Network and processor architecture for message-driven computers. In R. Sauya and G. Birtwistle, editors, *VLSI and Parallel Computation*. Morgan Kaufmann, San Mateo, CA, 1990.

[DeC89] A. L. DeCegama. *The Technology of Parallel Processing: Parallel Processing Architectures and VLSI Hardware: Volume 1*. Prentice-Hall, Englewood Cliffs, NJ, 1989.

[DS86] W. J. Dally and C. L. Seitz. The torus routing chip. *Journal of Distributed Computing*, 1(3):187–196, 1986.

[DS87] W. J. Dally and C. L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(5):547–553, 1987.

[Fen81] T. Y. Feng. A survey of interconnection networks. *IEEE Computer*, 12–27, December 1981.

[Fly72] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972.

[FW78] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of ACM Symposium on Theory of Computing*, 114–118, 1978.

[GGK+83] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer—designing a MIMD, shared memory parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, February 1983.

[Gib89] P. B. Gibbons. A more practical PRAM model. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, 158–168, 1989.

[GK93] A. Gupta and V. Kumar. The scalability of FFT on parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(7), July 1993. A detailed version available as Technical Report TR 90-53, Department of Computer Science, University of Minnesota, Minneapolis, MN.

[GR90] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, Cambridge, UK, 1990.

[HB84] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, New York, NY, 1984.

[HP89] S. W. Hornick and F. P. Preparata. Deterministic PRAM simulation with constant redundancy. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, 103–109, 1989.

[Hwa93] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, New York, NY, 1993.

[Jaj92] J. Jaja. *An Introduction to Parallel Algorithms*. Addison-Wesley, Reading, MA, 1992.

[Ken90] Kendall Square Research Corporation. *KSR-1 Overview*. Waltham, MA. 1990.

[KK79] P. Kermani and L. Kleinrock. Virtual cut-through: A new communication switching technique. *Computer Networks*, 3(4):267–286, 1979.

[KR87] V. K. P. Kumar and C. S. Raghavendra. Array processor with multiple broadcasting. *Journal of Parallel and Distributed Computing*, 173–190, 1987.

[KR88] R. M. Karp and V. Ramachandran. A survey of complexity of algorithms for shared-memory machines. Technical Report 408, University of California, Berkeley, 1988.

[KU86] A. R. Karlin and E. Upfal. Parallel hashing - an efficient implementation of shared memory. In *Proceedings of 18th ACM Conference on Theory of Computing*, 160–168, 1986.

[KV92] K. Kreeger and N. R. Vempaty. Comparison of meshes vs. hypercubes for data rearrangement. Technical Report UCF-CS-92-28, Department of Computer Science, University of Central Florida, Orlando, FL, 1992.

[Law75] D. H. Lawrie. Access and alignment of data in an array processor. *IEEE Transactions on Computers*, C-24(1):1145–1155, 1975.

[Lei85] C. E. Leiserson. Fat-trees : Universal networks for hardware efficient supercomputing. In *Proceedings of the 1985 International Conference on Parallel Processing*, 393–402, 1985.

[Lei92] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures*. Morgan Kaufmann, San Mateo, CA, 1992.

[Lil92] D. J. Lilja. *Architectural Alternatives for Exploiting Parallelism*. IEEE Computer Society Press, Los Alamitos, CA, 1992.

[LLG+92] D. Lenoski, J. Laudon, K. Gharachorloo, W. D. Weber, A. Gupta, J. L. Hennessy, M. Horowitz, and M. Lam. The stanford dash multiprocessor. *IEEE Computer*, 63–79, March 1992.

[LPP88] F. Luccio, A. Pietracaprina, and G. Pucci. A probabilistic simulation of PRAMs on a bounded degree network. *Information Processing Letters*, 28:141–147, July 1988.

[LPP89] F. Luccio, A. Pietracaprina, and G. Pucci. A new scheme for deterministic simulation of PRAMs in VLSI. *SIAM Journal of Computing*, 1989.

[LY86] M. Li and Y. Yesha. New lower bounds for parallel computations. In *Proceedings of 18th ACM Conference on Theory of Computing*, 177–187, 1986.

[MKRS88] R. Miller, V. K. P. Kumar, D. I. Reisis, and Q. F. Stout. Meshes with reconfigurable buses. In *Proceedings of MIT Conference on Advanced Research in VLSI*, 163–178, 1988.

[MV84] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21(4):339–374, November 1984.

[nCU90] nCUBE Corporation. *nCUBE 6400 Processor Manual*. Beaverton, OR. 1990.

[Ni91] L. M. Ni. A layered classification of parallel computers. In *Proceedings of 1991 International Conference for Young Computer Scientists*, 28–33, 1991.

[Nic90] J. R. Nickolls. The design of the MasPar MP-1: A cost-effective massively parallel computer. In *IEEE Digest of Papers—Comcom*, 25–28. IEEE Computer Society Press, Los Alamitos, CA, 1990.

[NM93] L. M. Ni and McKinley. A survey of wormhole routing techniques in direct connect networks. *IEEE Computer*, 26(2), February 1993.

[PBG+85] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norlton, and J. Weiss. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proceedings of 1985 International Conference on Parallel Processing*, 764–771, 1985.

[PY88] C. H. Papadimitriou and M. Yannakakis. Towards an architecture independent analysis of parallel algorithms. In *Proceedings of 20th ACM Symposium on Theory of Computing*, 510–513, 1988.

[RND77] E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, Englewood Cliffs, NJ, 1977.

[RS90] S. Ranka and S. Sahni. *Hypercube Algorithms for Image Processing and Pattern Recognition*. Springer-Verlag, New York, NY, 1990.

[SB77] H. Sullivan and T. R. Bashkow. A large scale, homogeneous, fully distributed parallel machine. In *Proceedings of Fourth Symposium on Computer Architecture*, 105–124, March 1977.

[Sei85] C. L. Seitz. The cosmic cube. *Communications of the ACM*, 28-1:22–33, 1985.

[Sei92] C. L. Seitz. Mosaic C: An experimental fine-grain multicomputer. Technical report, California Institute of Technology, Pasadena, CA, 1992.

[Sie85] H. J. Siegel. *Interconnection Networks for Large-Scale Parallel Processing*. D. C. Heath, Lexington, MA, 1985.

[SM86] S. J. Stolfo and D. P. Miranker. The DADO production system machine. *Journal of Parallel and Distributed Computing*, 3:269–296, June 1986.

[Sni82] M. Snir. On parallel search. In *Proceedings of Principles of Distributed Computing*, 242–253, 1982.

[Sni85] M. Snir. On parallel searching. *SIAM Journal of Computing*, 14(3):688–708, August 1985.

[Sny86] L. Snyder. Type architectures, shared-memory and the corollary of modest potential. *Annual Review of Computer Science*, 1:289–317, 1986.

[SS88] Y. Saad and M. H. Schultz. Topological properties of hypercubes. *IEEE Transactions on Computers*, 37:867–872, 1988.

[SS89a] Y. Saad and M. H. Schultz. Data communication in hypercubes. *Journal of Parallel and Distributed Computing*, 6:115–135, 1989. Also available as Technical Report YALEU/DCS/RR-428 from the Department of Computer Science, Yale University, New Haven, CT.

[SS89b] Y. Saad and M. H. Schultz. Data communication in parallel architectures. *Parallel Computing*, 11:131–150, 1989.

[Sto71] H. S. Stone. Parallel processing with the perfect shuffle. *IEEE Transactions on Computers*, C-20(2):153–161, 1971.

[Sto93] H. S. Stone. *High-Performance Computer Architecture: Third Edition*. Addison-Wesley, Reading, MA, 1993.

[Sup91] Supercomputer Systems Division, Intel Corporation. *Paragon XP/S Product Overview*. Beaverton, OR. 1991.

[Thi90] Thinking Machines Corporation. *The CM-2 Technical Summary*. Cambridge, MA. 1990.

[Thi91] Thinking Machines Corporation. *The CM-5 Technical Summary*. Cambridge, MA. 1991.

[Upf84] E. Upfal. A probabilistic relation between desirable and feasible models of parallel computation. In *Proceedings of 16th ACM Conference on Theory of Computing*, 258–265, 1984.

[UW84] E. Upfal and A. Widgerson. How to share memory in a distributed system. In *Proceedings of 25th Annual Symposium on the Foundation of Computer Science*, 171–180, 1984.

[Val90a] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), 1990.

[Val90b] L. G. Valiant. General purpose parallel architectures. *Handbook of Theoretical Computer Science*, 1990.

[WB72] W. A. Wulf and C. G. Bell. C.mmp—a multimicroprocessor. In *Proceedings of AFIPS Conference*, 765–777, 1972.

[WF80] C. L. Wu and T. Y. Feng. On a class of multistage interconnection networks. *IEEE Transactions on Computers*, 669–702, August 1980.