CHAPTER

12

LARGE COMPUTER SYSTEMS

CHAPTER OBJECTIVES

In this chapter you will learn about:

- Large computer systems that consist of multiple processors, or multiple computers
- Different structures for implementing multiprocessors
- Interconnection networks and LANs
- Memory organization in multiprocessors
- Cache coherence for shared data
- Shared-memory and message-passing paradigms
- Performance issues in multiprocessor systems

When a computer application requires a very large amount of computation to be completed in a reasonable amount of time, we must use machines with correspondingly large computing capacity. Such machines are often called *supercomputers*. Typical applications that require supercomputers include weather forecasting, finite element analysis in structural design, fluid flow analysis, simulation of large complex physical systems, and computer-aided design (CAD). None of the machines discussed in previous chapters are in the supercomputer class.

A high-performance processor can be designed using fast circuit technology and architectural features such as multiple functional units, pipelining, large caches, interleaved main memory, and separate buses for instructions and data. All these possibilities are exploited in ongoing research and development efforts by many manufacturers to produce processors intended primarily for use in workstations. Their quest is to increase performance without substantially increasing cost, and the results have been spectacular — we now have workstations that outperform machines that were considered to be in the supercomputer class only a decade ago.

However, many applications still demand computing power that greatly exceeds the capability of workstations. Thus, the need for supercomputing power remains strong. One approach is to build a supercomputer that has only a few very powerful processing units. This is typically done by using the fastest possible circuits, wide paths for accessing a large main memory, and extensive I/O capability. Such computers dissipate considerable power and require expensive cooling arrangements. In computationally demanding applications, supercomputers are needed to handle vectors of data, where a *vector* is a linear array of numbers (elements), as efficiently as possible. Single operations are often performed on entire vectors. For example, an add operation may generate a vector that is the element-by-element sum of two 64-element vectors. Also, a single memory access operation can cause an entire vector to be transferred between the main memory and processor registers. If an application is conducive to vector processing, then computers that feature a vector architecture provide excellent performance. Supercomputers of this class have been marketed by companies such as Cray (Cray-1, Y-MP, and SV1), Fujitsu (VP5000), Hitachi (SR8000), and NEC (SX-5). The main drawback of such machines has been their high cost — both the purchase price and the operating and maintenance cost.

An attractive alternative for providing supercomputing power is to use a large number of processors designed for the workstation market. This can be done in two basic ways. The first possibility is to build a machine that includes an efficient high-bandwidth medium for communication among the multiple processors, memory modules, and I/O devices. Such machines are usually referred to as *multiprocessors*. The second possibility is to implement a system using many workstations connected by a local area communication network. Systems of this type are often called *distributed computer systems*. Multiprocessors and distributed computer systems have many similarities. The former offer superior performance but at a higher price. The latter are naturally available in a modern computing environment at low cost. In the remainder of this chapter, we discuss the salient characteristics of each of these types. They provide large computing capabilities at a reasonable cost.

A system that uses many processors derives its high performance from the fact that many computations can proceed in parallel. The difficulty in using such a system

efficiently is that it may not be easy to break an application down into small tasks that can be assigned to individual processors for simultaneous execution. Determining these tasks and then scheduling and coordinating their execution in multiple processors requires sophisticated software and hardware techniques. We consider these issues later in the chapter.

12.1 FORMS OF PARALLEL PROCESSING

Many opportunities are available for parts of a given computational task to be executed in parallel. We have already seen several of them in earlier chapters. For example, in handling I/O operations, most computer systems have hardware that performs direct memory access (DMA) between an I/O device and main memory. The transfer of data in either direction between the main memory and a magnetic disk can be accomplished under the direction of a DMA controller that operates in parallel with the processor.

When a block of data is to be transferred from disk to main memory, the processor initiates the transfer by sending instructions to the DMA controller. While the controller transfers the required data using cycle stealing, the processor continues to perform some computation that is unrelated to the data transfer. When the controller completes the transfer, it sends an interrupt request to the processor to signal that the requested data are available in the main memory. In response, the processor switches to a computation that uses the data.

This simple example illustrates two fundamental aspects of parallel processing. First, the overall task has the property that some of its subtasks can be done in parallel by different hardware components. In this example, a processor computation and an I/O transfer are performed in parallel by the processor and the DMA controller. Second, some means must exist for initiating and coordinating the parallel activity. Initiation occurs when the processor sets up the DMA transfer and then continues with another computation. When the transfer is completed, the coordination is achieved by the interrupt signal sent from the DMA controller to the processor. This allows the processor to begin the computation that operates on the transferred data.

The preceding example illustrates a simple case of parallelism involving only two tasks. In general, large computations can be divided into many parts that can be performed in parallel. Several hardware structures can be used to support such parallel computations.

12.1.1 CLASSIFICATION OF PARALLEL STRUCTURES

A general classification of parallel processing has been proposed by Flynn [1]. In this classification, a single-processor computer system is called a *Single Instruction stream, Single Data stream* (SISD) system. A program executed by the processor constitutes the single instruction stream, and the sequence of data items that it operates on constitutes the single data stream. In the second scheme, a single stream of instructions is broadcast to a number of processors. Each processor operates on its own data. This scheme,

in which all processors execute the same program but operate on different data, is called a *Single Instruction stream, Multiple Data stream* (SIMD) system. The multiple data streams are the sequences of data items accessed by the individual processors in their own memories. The third scheme involves a number of independent processors, each executing a different program and accessing its own sequence of data items. Such machines are called *Multiple Instruction stream, Multiple Data stream* (MIMD) systems. The fourth possibility is a *Multiple Instruction stream, Single Data stream* (MISD) system. In such a system, a common data structure is manipulated by separate processors, each executing a different program. This form of computation does not occur often in practice, so it is not pursued here.

This chapter concentrates on MIMD structures because they are most useful for general purposes. However, we first briefly consider the SIMD structure to illustrate the kind of applications for which it is well-suited.

12.2 ARRAY PROCESSORS

The SIMD form of parallel processing, also called *array processing*, was the first form of parallel processing to be studied and implemented. In the early 1970s, a system named ILLIAC-IV [2] was designed at the University of Illinois using this approach and was later built by Burroughs Corporation. Figure 12.1 illustrates the structure of an array processor. A two-dimensional grid of processing elements executes an instruction stream that is *broadcast* from a central control processor. As each instruction is broadcast, all elements execute it simultaneously. Each processing element is connected to

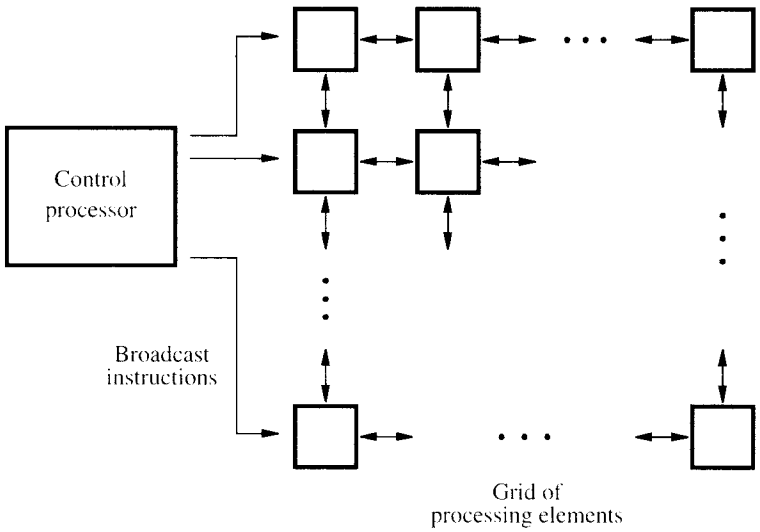


Figure 12.1 An array processor.

its four nearest neighbors for purposes of exchanging data. End-around connections may be provided in both rows and columns, but they are not shown in the figure.

Let us consider a specific computation in order to understand the capabilities of the SIMD architecture. The grid of processing elements can be used to solve two-dimensional problems. For example, if each element of the grid represents a point in space, the array can be used to compute the temperature at points in the interior of a conducting plane. Assume that the edges of the plane are held at some fixed temperatures. An approximate solution at the discrete points represented by the processing elements is derived as follows. The outer edges are initialized to the specified temperatures. All interior points are initialized to arbitrary values, not necessarily the same. Iterations are then executed in parallel at each element. Each iteration consists of calculating an improved estimate of the temperature at a point by averaging the current values of its four nearest neighbors. The process stops when changes in the estimates during successive iterations are less than some predefined small quantity.

The capability needed in the array processor to perform such calculations is quite simple. Each element must be able to exchange values with each of its neighbors over the paths shown in the figure. Each processing element has a few registers and some local memory to store data. It also has a register, which we can call the network register, that facilitates movement of values to and from its neighbors. The central processor can broadcast an instruction to shift the values in the network registers one step up, down, left, or right. Each processing element also contains an ALU to execute arithmetic instructions broadcast by the control processor. Using these basic facilities, a sequence of instructions can be broadcast repeatedly to implement the iterative loop. The control processor must be able to determine when each of the processing elements has developed its component of the temperature to the required accuracy. To do this, each element sets an internal status bit to 1 to indicate this condition. The grid interconnections include a facility that allows the controller to detect when all status bits are set at the end of an iteration.

An interesting question with respect to array processors is whether it is better to use a relatively small number of powerful processors or a large number of very simple processors. ILLIAC-IV is an example of the former choice. Its 64 processors had a 64-bit internal structure. Array processors introduced in the late 1980s are examples of the latter choice. The CM-2 machine produced by the Thinking Machines Corporation could accommodate up to 65,536 processors, but each processor is only one bit wide. Maspar's MP-1216 has a maximum of 16,384 processors that are 4 bits wide. The Cambridge Parallel Processing Gamma II Plus machines can have up to 4096 processors that can operate on either byte-sized or bit-sized operands. These choices reflect the belief that, in the SIMD environment, it is more useful to have a high degree of parallelism rather than to have fewer but more powerful processors.

Array processors are highly specialized machines. They are well-suited to numerical problems that can be expressed in matrix or vector format. Recall that supercomputers with a vector architecture are also suitable for solving such problems. A key difference between vector-based machines and array processors is that the former achieve high performance through heavy use of pipelining, whereas the latter provide extensive parallelism by replication of computing modules. Neither array processors nor vector-based machines are particularly useful in speeding up general computations, and they do not have a large commercial market.

12.3 THE STRUCTURE OF GENERAL-PURPOSE MULTIPROCESSORS

The array processor architecture described in the preceding section is a design for a computer system that corresponds directly to a class of computational problems that exhibit an obvious form of data parallelism. In more general cases in which parallelism is not so obvious, it is useful to have an MIMD architecture, which involves a number of processors capable of independently executing different routines in parallel.

Figures 12.2, 12.3, and 12.4 show three possible ways of implementing a multiprocessor system. The most obvious scheme is given in Figure 12.2. An *interconnection network* permits n processors to access k memories so that any of the processors can access any of the memories. The interconnection network may introduce considerable delay between a processor and a memory. If this delay is the same for all accesses to memory, which is common for this organization, then such a machine is called

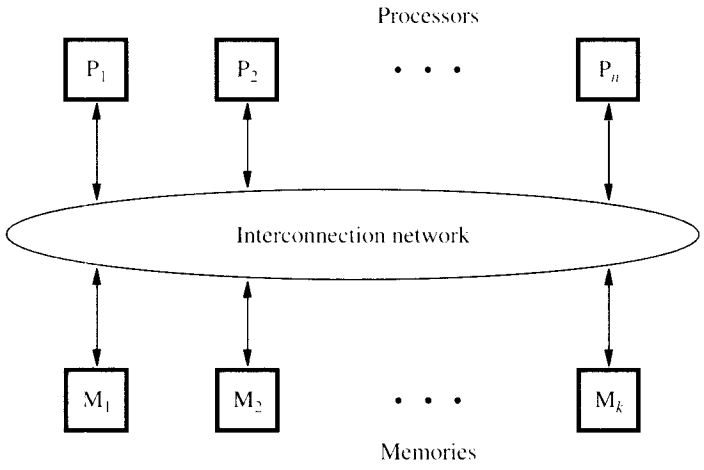


Figure 12.2 A UMA multiprocessor.

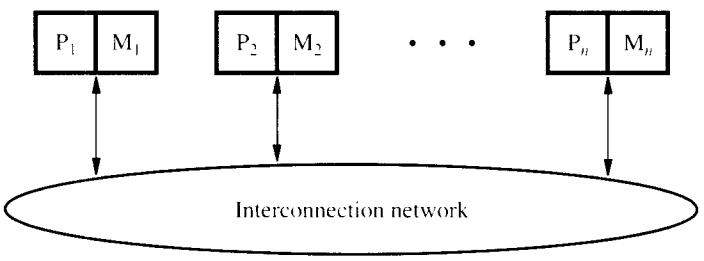


Figure 12.3 A NUMA multiprocessor.

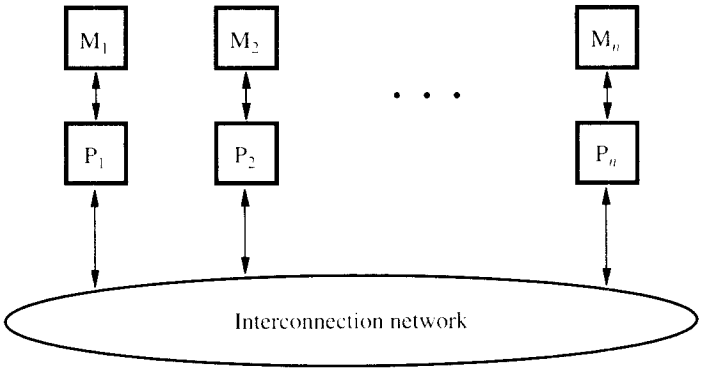


Figure 12.4 A distributed memory system.

a Uniform Memory Access (UMA) multiprocessor. Because of the extremely short instruction execution times achievable by processors, the network delay in fetching instructions and data from the memories is unacceptable if it is too long. Unfortunately, interconnection networks with very short delays are costly and complex to implement.

An attractive alternative, which allows a high computation rate to be sustained in all processors, is to attach the memory modules directly to the processors. This organization is shown in Figure 12.3. In addition to accessing its local memory, each processor can also access other memories over the network. Since the remote accesses pass through the network, these accesses take considerably longer than accesses to the local memory. Because of this difference in access times, such multiprocessors are called Non-Uniform Memory Access (NUMA) multiprocessors.

The organizations of Figures 12.2 and 12.3 provide a *global memory*, where any processor can access any memory module without intervention by another processor. A different way of organizing the system is shown in Figure 12.4. Here, all memory modules serve as private memories for the processors that are directly connected to them. A processor cannot access a remote memory without the cooperation of the remote processor. This cooperation takes place in the form of messages exchanged by the processors. Such systems are often called *distributed-memory* systems with a *message-passing protocol*.

The preceding discussion uses processors and memory modules as the main functional units in a multiprocessor system. Although we have not discussed I/O modules explicitly, any multiprocessor must provide extensive I/O capability. This capability can be provided in different ways. Separate I/O modules can be connected directly to the network, providing standard I/O interfaces, as discussed in Chapter 4. Some I/O functions can also be incorporated into the processor modules.

Figures 12.2, 12.3, and 12.4 depict a high-level view of possible multiprocessor organizations. The performance and cost of these machines depend greatly on implementation details. In the next two sections, we consider the most popular schemes for realizing the communication network and the structure of the memory hierarchy.

12.4 INTERCONNECTION NETWORKS

In this section, we examine some of the possibilities for implementing the interconnection network in multiprocessor systems. In general, the network must allow information transfer between any pair of modules in the system. The network may also be used to broadcast information from one module to many other modules. The traffic in the network consists of requests (such as read and write), data transfers, and various commands.

The suitability of a particular network is judged in terms of cost, bandwidth, effective throughput, and ease of implementation. The term *bandwidth* refers to the capacity of a transmission link to transfer data and is expressed in bits or bytes per second. The *effective throughput* is the actual rate of data transfer. This rate is less than the available bandwidth because a given link usually does not carry data all of the time.

Information transfer through the network usually takes place in the form of *packets* of fixed length and specified format. For example, a read request is likely to be a single packet that contains the addresses of the source (the processor module) and destination (the memory module) and a command field that indicates what type of read operation is required. A write request that writes one word in a memory module is also likely to be a single packet that includes the data to be written. On the other hand, a read response that involves an entire cache block requires several packets. Longer messages may require many packets.

Ideally, a complete packet would be handled in parallel in one clock cycle at any node or switch in the network. This implies having wide links, comprising many wires. However, to reduce cost and complexity, the links are often considerably narrower. In such cases, a packet must be divided into smaller pieces, each of which can be transmitted in one clock cycle.

12.4.1 SINGLE BUS

The simplest and most economical means for interconnecting a number of modules is to use a single bus. The detailed aspects of bus design, as discussed in Chapter 4, apply here as well. Since several modules are connected to the bus and any module can request a data transfer at any time, it is essential to have an efficient bus arbitration scheme. Examples of such schemes are given in Chapter 4.

In a simple mode of operation, the bus is dedicated to a particular source-destination pair for the full duration of the requested transfer. For example, when a processor issues a read request on the bus, it holds the bus until it receives the desired data from the memory module. Since the memory module needs a certain amount of time to access the data (as discussed in Chapter 5), the bus will be idle until the memory is ready to respond with the data. Then the data are transferred to the processor. When this transfer is completed, the bus can be assigned to handle another request.

Suppose that a bus transfer takes T time units, and the memory access time is $4T$ units. It then takes $6T$ units to complete a read request. Thus, the bus is idle for two-thirds of the time. A scheme known as the *split-transaction protocol* makes it possible to use the bus during the idle period to serve another request. Consider the following

method of handling a series of read requests, possibly from different processors. After transferring the address involved in the first request, the bus may be reassigned to transfer the address for the second request. Assuming that this request is to a different memory module, we now have two modules proceeding with read access cycles in parallel. If neither module has finished with its access, the bus may be reassigned to a third request, and so on. Eventually, the first memory module completes its access cycle and uses the bus to transfer the word to the source that requested it. As other modules complete their cycles, the bus is used to transfer their data to the corresponding sources. Note that the actual length of time between address transfer and word return is not critical. Address and data transfers for different requests represent independent uses of the bus that can be interleaved in any order.

The split-transaction protocol allows the bus and the available bandwidth to be used more efficiently. The performance improvement achieved with this protocol depends on the relationship between the bus transfer time and the memory access time. Performance is improved at the cost of increased bus complexity. There are two reasons why complexity increases. Since a memory module needs to know which source initiated a given read request, a source identification tag must be attached to the request. This tag is later used to send the requested data to the source. Complexity also increases because all modules, not just the processors, must be able to act as bus masters.

Multiprocessors that use the split-transaction bus vary in size from 4 to 32 processors. In larger sizes, the bandwidth of the bus can become a problem. The bandwidth can be increased if a wider bus, that is, a bus that has more wires, is used. Most of the data transferred between processors and memory modules consist of cache blocks, where a block consists of a number of words. If the bus is wide enough to transfer several words at a time, then a complete block can be transferred more quickly than if the words are transferred one at a time. The Challenge multiprocessor from Silicon Graphics Corporation uses a bus that allows parallel transfer of 256 bits of data.

The main limitation of a single bus is that the number of modules that can be connected to the bus is not large. An ordinary bus functions well if no more than 10 to 15 modules are connected to it. Using a wider bus to increase the bandwidth allows the number of modules to be doubled. The bandwidth of a single bus is limited by contention for the use of the bus and by the increased propagation delays caused by electrical loading when many modules are connected. Networks that allow multiple independent transfer operations to proceed in parallel can provide significantly increased data transfer rates.

12.4.2 CROSSBAR NETWORKS

A versatile switching arrangement is shown in Figure 12.5. It is known as the *crossbar switch*, which was originally developed for use in telephone networks. For clarity of illustration, the switches in the figure are depicted as mechanical switches, although in practice these are electronic switches. Any module, Q_i, can be connected to any other module, Q_j, by closing an appropriate switch. Such networks, where there is a direct link between all pairs of nodes, are called *fully connected* networks. Many simultaneous transfers are possible. If n sources need to send data to n distinct destinations, then all

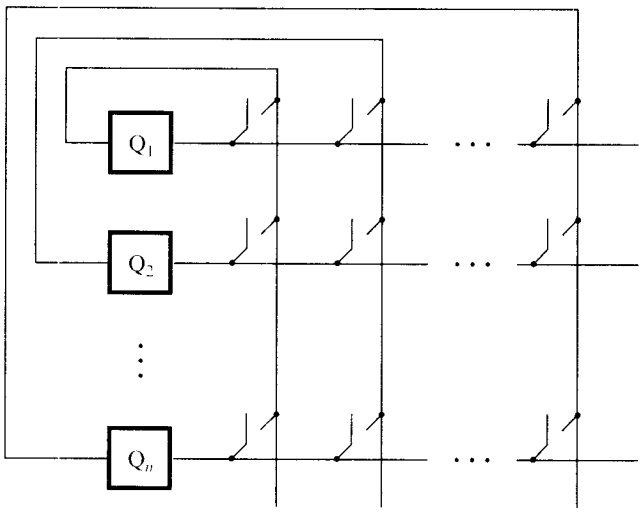


Figure 12.5 Crossbar interconnection network.

of these transfers can take place concurrently. Since no transfer is prevented by the lack of a communication path, the crossbar is called a *nonblocking switch*.

In Figure 12.5, we show just a single switch at each crosspoint. In an actual multiprocessor, however, the paths through the crossbar network are much wider. This means that many switches are needed at each crosspoint. Since the number of crosspoints is n^2 in a network used to interconnect n modules, the total number of switches becomes large as n increases. This results in high cost and cumbersome implementation. Crossbars are attractive as interconnection networks when the number of interconnected nodes is not large.

One of the larger crossbar switches is found in Sun's E10000 system, in which 16 four-processor nodes are connected by a 16 × 16 crossbar switch. It is also possible to use a multilevel crossbar switch, where a crossbar switch at level 1 connects to a crossbar switch at level 2, and so on. In this way it is possible to connect a larger number of processors. Such schemes are found in Fujitsu's VPP5000, Hitachi's SR8000, and NEC's SX-5 machines. A multilevel crossbar has become a popular choice for a high-performance interconnection medium.

12.4.3 MULTISTAGE NETWORKS

The bus and crossbar systems just described use a single stage of switching to provide a path from a source to a destination. It is also possible to implement interconnection networks that use multiple stages of switches to set up paths between sources and destinations. Such networks are less costly than the crossbar structure, yet they provide a reasonably large number of parallel paths between sources and destinations. Multistage switching is best illustrated by an example. Figure 12.6 shows a three-stage

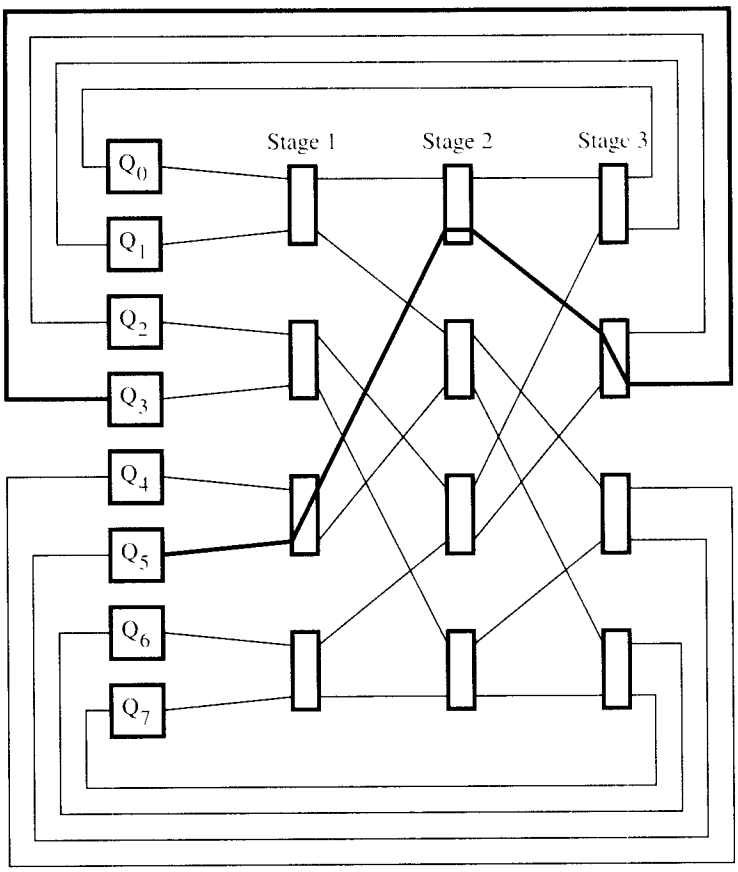


Figure 12.6 Multistage shuffle network.

network called a *shuffle network* that interconnects eight modules. The term "shuffle" describes the pattern of connections from the outputs of one stage to the inputs of the next stage. This pattern is identical to the repositioning of playing cards in a deck that is shuffled by splitting the deck into two halves and interleaving the cards in each half.

Each switchbox in the figure is a 2×2 switch that can route either input to either output. If the inputs request distinct outputs, then they can both be routed simultaneously in the straight-through or crossed pattern. If both inputs request the same output, only one request can be satisfied. The other one is blocked until the first request finishes using the switch. It can be shown that a network consisting of s stages can be used to interconnect 2^s modules. In this case, there is exactly one path through the network from any module Q_i to any other module Q_j. Therefore, this network provides full connectivity between sources and destinations. Many request patterns, however, cannot be satisfied simultaneously. For example, the connection from Q_0 to Q_4 cannot be provided at the same time as the connection from Q_1 to Q_5.

A multistage network is less expensive to implement than a crossbar network. If n nodes are to be interconnected using the scheme in Figure 12.6, then we must use $s = \log_2 n$ stages with $n/2$ switchboxes per stage. Since each switchbox contains four switches, the total number of switches is

$$4 \times \frac{n}{2} \times \log_2 n = 2n \times \log_2 n$$

which, for large networks, is considerably less than the n^2 switches needed in a crossbar network.

A particular request can be routed through the network using the following scheme. The source sends a binary pattern representing the destination number into the network. As the pattern moves through the network, each stage examines a different bit to determine switch settings. Stage 1 uses the most significant bit, stage 2 the middle bit, and stage 3 the least significant bit. When a request arrives on either input of a switch, it is routed to the upper output if the controlling bit is a 0 and to the lower output if the controlling bit is a 1. For example, a request from source Q_5 for destination Q_3 moves through the network as shown by the blue lines in Figure 12.6. Its route is controlled by the bit pattern 011, which is the destination address.

A good example of a multiprocessor based on a multistage network was the BBN Butterfly manufactured by BBN Advanced Computers. A 64-processor model of this system contained a three-stage network built with 4×4 switches. The routing through each stage of these switches was determined by successive 2-bit fields of the destination address. A current example is the IBM RS/6000 SP multiprocessor, which can use a multistage network as one of several options for interconnecting clusters of processors.

Multistage networks are less capable of providing concurrent connections than crossbar switches, but they are also less costly to implement. Interest in these networks peaked in the 1980s and has diminished greatly in the past few years. Other schemes, which we discuss in the remainder of this section, have become more attractive.

12.4.4 HYPERCUBE NETWORKS

In the three schemes discussed previously, the interconnection network imposes the same delay for paths connecting any two modules. Such schemes can be used to implement UMA multiprocessors. We now discuss network topologies that are suitable only for NUMA multiprocessors. The first such scheme that gained popularity uses the topology of an n-dimensional cube, called a *hypercube*, to implement a network that interconnects 2^n nodes. In addition to the communication circuits, each node usually includes a processor and a memory module as well as some I/O capability.

Figure 12.7 shows a three-dimensional hypercube. The small circles represent the communication circuits in the nodes. The functional units attached to each node are not shown in the figure. The edges of the cube represent bidirectional communication links between neighboring nodes. In an n-dimensional hypercube, each node is directly connected to n neighbors. A useful way to label the nodes is to assign binary addresses

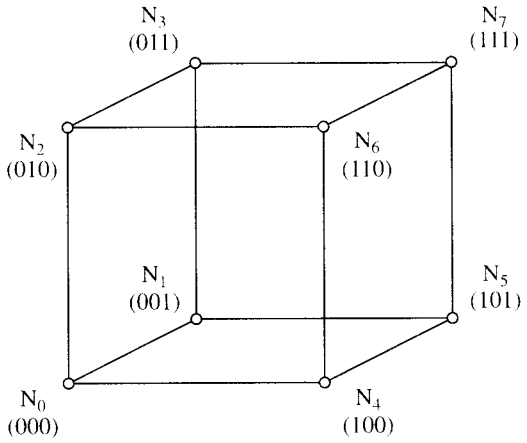


Figure 12.7 A 3-dimensional hypercube network.

to them in such a way that the addresses of any two neighbors differ in exactly one bit position, as shown in the figure.

Routing messages through the hypercube is particularly easy. If the processor at node N_i wishes to send a message to node N_j, it proceeds as follows. The binary addresses of the source, i, and the destination, j, are compared from least to most significant bits. Suppose that they differ first in position p. Node N_i then sends the message to its neighbor whose address, k, differs from i in bit position p. Node N_k forwards the message to the appropriate neighbor using the same address comparison scheme. The message gets closer to destination node N_j with each of these hops from one node to another. For example, a message from node N_2 to node N_5 requires 3 hops, passing through nodes N_3 and N_1. The maximum distance that any message needs to travel in an n-dimensional hypercube is n hops.

Scanning address patterns from right to left is only one of the methods that can be used to determine message routing. Any other scheme that moves a message closer to its destination on each hop is equally acceptable, as long as the routing decision can be made at each node on the path using only local information. This feature of the hypercube is attractive from the reliability viewpoint. The existence of multiple paths between two nodes means that when faulty links are encountered, they can usually be avoided by simple, local routing decisions. If one of the shortest routes is not available, a message may be sent over a longer path. When this is done, care must be taken to avoid looping, which is the situation in which the message circulates in a closed loop and never reaches its destination.

Hypercube interconnection networks have been used in a number of machines. The better known examples include Intel's iPSC, which used a 7-dimensional cube to connect up to 128 nodes, and NCUBE's NCUBE/ten, which had up to 1024 nodes in a 10-dimensional cube. The hypercube networks lost much of their popularity in the early 1990s when mesh-based structures emerged as a more attractive alternative.

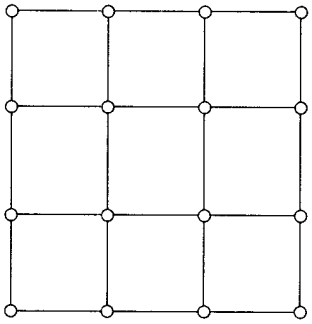


Figure 12.8 A 2-dimensional mesh network.

12.4.5 MESH NETWORKS

One of the most natural ways of interconnecting a large number of nodes is by means of a *mesh*. An example of a mesh with 16 nodes is given in Figure 12.8. Again, the links between the nodes are bidirectional. Meshes gained popularity in the early 1990s and essentially displaced hypercubes as a choice for interconnection networks in large multiprocessors.

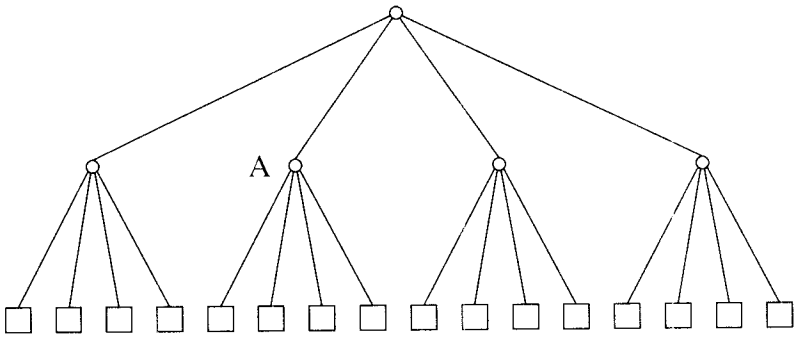
Routing in a mesh network can be done in several different ways. One of the simplest and most effective possibilities is to choose the path between a source node N_i and a destination node N_j such that the transfer first takes place in the horizontal direction from N_i toward N_j. When the column in which N_j resides is reached, the transfer proceeds in the vertical direction along this column. Well-known examples of mesh-based multiprocessors are Intel's Paragon and the experimental machines Dash [3] and Flash [4] at Stanford University and Alewife [5] at MIT.

If a wraparound connection is made between the nodes at the opposite edges in Figure 12.8, the result is a network that consists of a set of bidirectional rings in the X direction connected by a similar set of rings in the Y direction. In this network, called a *torus*, the average latency of information transfer is reduced, but at the cost of greater complexity. Such an interconnection network is used in Fujitsu's AP3000 machines.

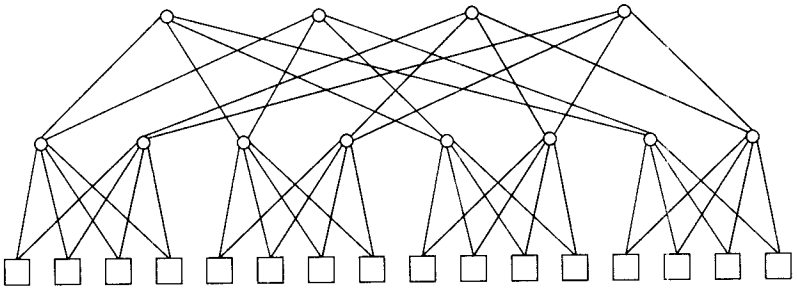
Both the regular mesh and the torus schemes can also be implemented as three-dimensional networks, in which the links are between neighbors in the X, Y, and Z directions. An example of a three-dimensional torus is found in Cray's T3E multiprocessor.

12.4.6 TREE NETWORKS

A hierarchically structured network implemented in the form of a tree is another interconnection topology. Figure 12.9*a* depicts a four-way tree that interconnects 16 modules. In this tree, each parent node allows communication between two of its children at a time. An intermediate-level node, for example node A in the figure, can provide a connection from one of its child nodes to its parent. This enables two leaf nodes that are



(a) Four-way tree



(b) Fat tree

Figure 12.9 Tree-based networks.

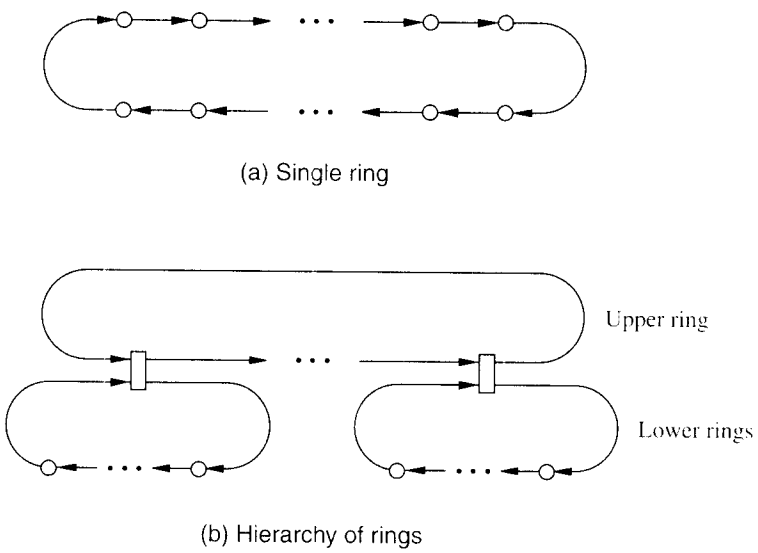
any distance apart to communicate. Only one path at a time can be established through a given node in the tree.

A tree network performs well if there is a large amount of locality in communication, that is, if only a small portion of network traffic goes through the single root node. If this is not the case, performance deteriorates rapidly because the root node becomes a bottleneck.

To reduce the possibility of a bottleneck, the number of links in the upper levels of a tree hierarchy can be increased. This is done in a *fat tree* network, in which each node in the tree (except at the top level) has more than one parent. An example of a fat tree is given in Figure 12.9*b*. In this case, each node has two parent nodes. A fat tree structure was used in the CM-5 machine by Thinking Machines Corporation.

12.4.7 RING NETWORKS

One of the simplest network topologies uses a ring to interconnect the nodes in the system, as shown in Figure 12.10*a*. The main advantage of this arrangement is that the ring is easy to implement. Links in the ring can be wide, usually accommodating



(a) Single ring

(b) Hierarchy of rings

Figure 12.10 A ring-based interconnection network.

a complete packet in parallel, because each node is connected to only two neighbors. However, it is not useful to construct a very long ring to connect many nodes because the latency of information transfer would be unacceptably large.

Rings can be used as building blocks for the topologies discussed in previous sections, such as meshes, hypercubes, trees, and fat trees. We consider the simple possibility of using rings in a tree structure; this results in a hierarchy of rings as shown in Figure 12.10*b*. A two-level hierarchy is depicted in the figure, but more levels can be used. Having short rings reduces substantially the latency of transfers that involve nodes on the same ring. Moreover, the latency of transfers between two nodes on different rings is shorter than if a single ring were used. The drawback of this scheme is that the highest-level ring may become a bottleneck for traffic.

Commercial machines that feature ring networks include Exemplar V2600 by Hewlett-Packard and KSR-2 by Kendal Square Research. Rings have also been used in the experimental machines Hector [6] and NUMAchine [7] at the University of Toronto.

12.4.8 PRACTICAL CONSIDERATIONS

We have seen that several different topologies can be used to implement the interconnection network in a multiprocessor system. It would be difficult to argue that any topology is clearly superior to others. Each has certain advantages and disadvantages. When comparing different approaches, we must take into account several practical considerations.

The most fundamental requirement is that the communication network be fast enough and have sufficient throughput to satisfy the traffic demand in a multiprocessor

system. This implies high speed of transfer along the communication path and a simple routing mechanism to allow routing decisions to be made quickly. The network should be easy to implement; the wiring complexity must be reasonable and conducive to simple packaging. Complexity is inevitably reflected in the cost of the network, which is another major consideration.

Multiprocessors of different sizes are needed. The ideal network would be suitable for all sizes, ranging from just a few processors to possibly thousands of processors. The term *scalability* is often used to describe the ability of a multiprocessor architecture (which includes the interconnection network) to provide increased performance as the size of the system increases, while the increase in cost is proportional to the increase in size. It is particularly advantageous if a relatively small multiprocessor system can be acquired at a low cost but can be easily expanded to a large system with a linear increase in cost and performance. Unfortunately, this is not true for many commercial products. Often, the up-front cost for even a small system is large because much of the communication hardware needed to accommodate a larger system must be provided in one piece.

In addition to providing the basic communication between sources and destinations, it is useful to have *broadcasting* capability where a message traverses the entire network and is received by all nodes. The ability to send a message to only a subset of the network nodes is also beneficial. Such transfers are called *multicasting*.

The choice of the interconnection network affects the implementation of schemes used to ensure that any multiple copies of data that may exist in caches of different processors acquire the updates made so that all copies always have the same values. Such schemes are discussed in Section 12.6.2.

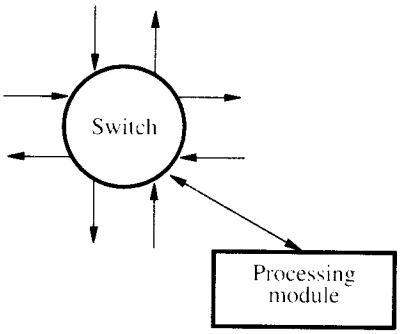
Reliability is another important factor. The more complex the network, the more likely it is to fail. Ideally, the machine could continue to function even if some link in the network fails. This is possible in networks that provide at least two different paths between each pair of communicating nodes. In general, simple networks tend to be robust, and they do not fail any more often than the processing and memory modules in the system. Highly reliable networks that include additional hardware can be built at considerable cost. This topic is beyond the scope of this book.

To demonstrate how all these characteristics can be evaluated, let us make a brief qualitative comparison of networks based on meshes and rings.

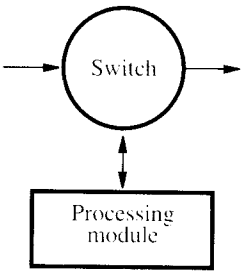
Meshes and Rings

Both mesh and ring networks are characterized by point-to-point links (connecting adjacent nodes), which can be driven at high clock rates. Both are viable in small configurations and can be expanded without difficulty. Incremental expansion is simpler in a ring network than in a mesh network.

In Figures 12.8 and 12.10, we indicate the nodes in the network as small circles and the links as single lines. Consider a more detailed picture; Figure 12.11 shows the communication paths associated with one node that has a processing module attached to it. The switch block includes both the circuitry that selects the path for a transfer and the buffers needed to hold the data being transferred. Data are transferred from the buffer in one node to the buffer in the next node in one clock cycle. Figure 12.11*a* depicts a node in a two-dimensional mesh network. Since bidirectional communication is needed



(a) Node in a mesh



(b) Node in a ring

Figure 12.11 Nodes in mesh and ring networks.

in both the X and Y directions, eight distinct network links must be connected to the node. The width of these links is limited by the total number of wires that can be used, taking into account the cost and packaging. Thus, it is unlikely that an individual link could be wide enough to carry an entire packet in parallel. To deal with this constraint, a packet can be divided into smaller portions to correspond to the width of the link. The term *flit* (FLow control digIT) is often used to refer to a portion of the packet that can be accepted by the switching circuitry in the node for forwarding, or buffering in case the forward path is blocked by another transfer. In practice, it is convenient if the flit corresponds to the width of the link.

If a packet must be divided into flits, how should it be routed through the network? A straightforward scheme, known as the *store-and-forward* method, is to provide a large enough buffer in each node to hold all flits of a packet. Thus, an entire packet is transferred from one node to another, where it is stored until it can be forwarded to the next node. (The number of clock cycles required for the transfer depends on the number of flits.) The negative aspects of this scheme are the size of the buffers needed and the increased latency in passing through a node. An attractive alternative

is the *wormhole* routing scheme (which has also been referred to as *pipelining*), in which the sequence of flits that constitute a packet can be viewed as a *worm* that moves through the network. The first flit in a worm contains a *header* that includes the address of the destination node. As this flit moves through the network, it establishes a path along which the remaining flits will pass. The tail of the worm closes the established path. The head of the worm may be temporarily blocked at any node, because another worm may be passing through this node. However, once the head moves, the rest of the worm moves along in subsequent clock cycles. Some control mechanism must stop the transmission of flits from preceding nodes when the head of the worm is blocked; a simple scheme using two buffers per node for each direction of transfer has been developed for this purpose [8]. Wormhole routing has lower latency than store-and-forward routing because the head flit is sent on its way without waiting for the remaining flits of the packet.

Wormhole routing is an application of a strategy known as *circuit switching*, which is a familiar concept from telephone networks, where a path through the network is established when a number is dialed. The conversation takes place along this path, called a *circuit*. The circuit is deactivated when the calling party hangs up. In the case of wormhole routing, it is the head flit that establishes the path. The progression of this flit may be temporarily blocked as explained above. Once a circuit is established, however, the remaining flits of the packet move toward the destination without experiencing any contention. In contrast, strategies in which an entire packet is buffered at each node, as in the store-and-forward method, are called *packet switching*. In this case, no circuit is set up, and the packet moves through the network as the buffer in each node becomes available.

Connections to a node in a ring network are shown in Figure 12.11*b*. Here, transfer occurs in only one direction, in addition to the connection to the processing module. Thus, the width of the link can be four times that in a mesh network for the same wire count. This means that it is feasible for an entire packet to be transferred in parallel from one node to another in one clock cycle. Figure 12.11*b* shows a node in the lowest level ring, to which a processing module is attached. If a ring hierarchy such as that in Figure 12.10*b* is used, then the inter-ring interfaces between lower and upper rings will have two input and two output links, one of each belonging to the upper- and lower-level rings.

Routing in a hierarchical ring network is very simple. A packet is never blocked, except possibly at an inter-ring interface when incoming packets on both upper- and lower-level rings are destined to continue along just one of the rings. To handle this situation, buffers (queues) must be provided in the interface, one from the lower- to the upper-level ring and another in the opposite direction. A processing module may inject a new packet onto the ring whenever no packet is arriving to the node from its upstream neighbor.

Next we consider the ability of networks to broadcast or multicast data. This ability is naturally available in ring networks. For example, a packet can be broadcast to all nodes by sending it to the top-level ring. As the packet traverses this ring, a copy is made at each inter-ring interface and sent along to the next lower-level ring. This process is repeated at all levels so that the copies of the original packet visit all nodes in the lowest-level rings. Broadcasting in a mesh network is more difficult, because the broadcast packet has to be broken up into flits and the progress of the broadcast

worm may be blocked at various nodes by other traffic. Moreover, the completion of a broadcast is not easy to detect.

The main disadvantage of a hierarchical ring network is that the ring at the top of the hierarchy may become a bottleneck if too many packets need to be transferred over it. This will occur if the locality in communication is low. The limited bandwidth of the top-level ring restricts the scalability of systems based on such networks to hundreds of processors. In contrast, mesh-based systems scale well to thousands of processors.

The preceding discussion shows that both meshes and rings are good choices for interconnection networks. Ring-based systems are easier to implement, but do not scale as well as mesh-based systems. Thus, rings merit serious consideration if the maximum size of the system is a few hundred processors. Mesh systems are suitable for use in both small and very large systems. For very small systems, say, up to 16 processors, the most effective choices are a single bus or a crossbar switch.

Since the size of a multiprocessor system has important implications, the reader may wonder what range of systems are in practical use. Most multiprocessor systems are relatively small. Many machines are in the range of 4 to 128 processors. Some very large machines with thousands of processors exist. However, the market for such large machines is small.

12.4.9 MIXED TOPOLOGY NETWORKS

We have considered several possible network topologies and showed that all existing topologies have certain advantages and disadvantages. Designers of multiprocessor systems strive to achieve superior performance at a reasonable cost. In an effort to exploit the most advantageous characteristics of different topologies, many successful machines feature mixed topologies. Bus and crossbar are excellent choices for connecting a few processors together. So, we often see a cluster of processors, typically from 2 to 8, connected using a bus or a crossbar. Such clusters, usually referred to as nodes, are then interconnected using a suitable topology to form a larger system.

Data General's AV25000 system uses nodes where processors are connected by a bus. These nodes are then interconnected using a ring network. Hewlett-Packard's Exemplar V2600 also uses a ring network to interconnect nodes, where each node has a crossbar switch connecting the processors. Compaq's AlphaServer SC uses a fat tree to interconnect the nodes that comprise processors connected by a crossbar switch.

12.4.10 SYMMETRIC MULTIPROCESSORS

Consider a multiprocessor system in which all processors have identical access to all memory modules and all I/O devices, so that the operating system software can treat any processor as interchangeable with any other processor. Then, if any processor can execute either the operating system kernel or user programs, the machine is called a *symmetric multiprocessor* (SMP). This also implies that any processor can initiate an I/O operation on any I/O device, and it can handle any external interrupt.

SMPs are usually implemented using either a bus or a crossbar network. Often, an SMP is used as a node in a much larger multiprocessor system. For example, SMP nodes are used in the Exemplar V2600 and AlphaServer SC multiprocessors mentioned above.

12.5 MEMORY ORGANIZATION IN MULTIPROCESSORS

In Chapter 5 we saw that the organization of the memory in a uniprocessor system has a large impact on performance. The same is true in multiprocessor systems. To exploit the locality of reference phenomenon, each processor usually includes a primary cache and a secondary cache. If the organization in Figure 12.2 is used, then each processor module can be connected to the communication network as shown in Figure 12.12. Only the secondary cache is shown in the figure since the primary cache is assumed to be a part of the processor chip. The memory modules are accessed using a single *global address space*, where a range of physical addresses is assigned to each memory module. In such a *shared memory* system, the processors access all memory modules in the same way. From the software standpoint, this is the simplest use of the address space.

In NUMA-organized multiprocessors, shown in Figure 12.3, each node contains a processor and a portion of the memory. A natural way of implementing the node is illustrated in Figure 12.13. In this case, it is also convenient to use a single global address space. Again, the processor accesses all memory modules in the same way, but

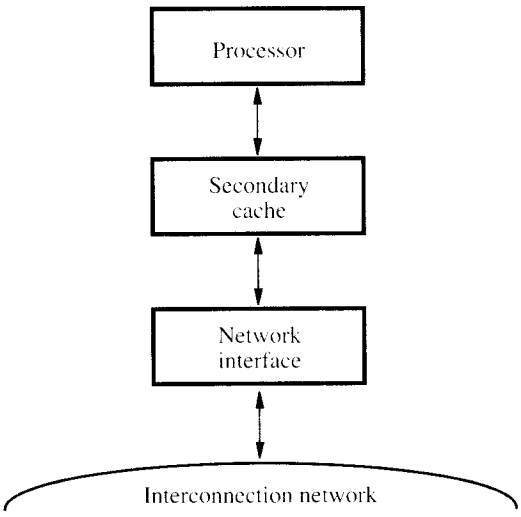


Figure 12.12 A processor node for the multiprocessor organization in Figure 12.2.

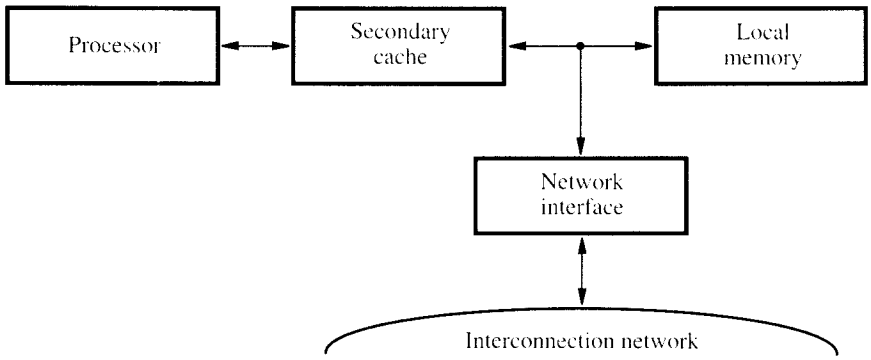


Figure 12.13 Node structure for the multiprocessor organization in Figure 12.3.

the accesses to the local memory component of the global address space take less time to complete than accesses to remote memory modules.

In the organization of Figure 12.4, each processor accesses directly only its own local memory. Thus, each memory module constitutes the *private address space* of one processor; there is no global address space. Any interaction among programs or processes running on different processors is implemented by sending *messages* from one processor to another. In this form of communication, each processor views the interconnection network as an I/O device. In effect, each node in such a system behaves as a computer in the same manner as discussed in previous chapters for uniprocessor machines. For this reason, systems of this type are referred to also as *multicomputers*. This organization provides the easiest way to connect a number of computers into a large system. Communication between tasks running on different computers is relatively slow because the exchange of messages requires software intervention. We consider this type of system in Section 12.7.

When data are shared among many processors, we must ensure that the processors observe the same value for a given data item. The presence of many caches in a shared-memory system creates a problem in this respect. Multiple copies of some data items may exist in various caches. Whenever a processor changes (writes) a data item in its own cache, the same change must be made in all caches that have a copy. Alternatively, the other copies must be invalidated. In other words, shared data must be *coherent* in all caches in the system. The problem of maintaining cache coherence can be solved in several different ways. We examine the most popular solutions in Section 12.6.2.

12.6 PROGRAM PARALLELISM AND SHARED VARIABLES

The introduction to this chapter states that it is difficult to break large tasks down into subtasks that can be executed in parallel on a multiprocessor. In some special cases, however, this division is easy. If a large task originates as a set of independent programs, then these programs can simply be executed on different processors. Unless

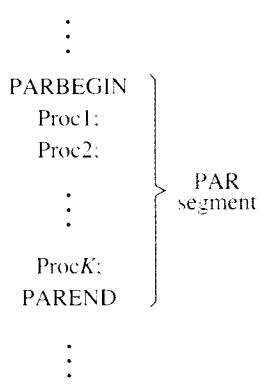


Figure 12.14 Parallel programming construct.

these programs block each other in competing for shared I/O devices, the multiprocessor is fully used by such a workload.

Another easy case occurs when a high-level source programming language has constructs that allow an application programmer to explicitly declare that certain subtasks of a program can be executed in parallel. Figure 12.14 shows such a construct, often called a PAR segment. The PARBEGIN and PAREND control statements bracket a list of procedures, named Proc1 through Proc$K$, that can be executed in parallel. The order of execution of this program is as follows. When the segment of the program preceding the PARBEGIN statement is completed, any or all of the K parallel procedures can be started immediately, depending on the number of idle processors available. They can be started in any order. Execution of the part of the program following PAREND is allowed to begin only after all of the K procedures have completed execution.

If this program is the only one being executed on the multiprocessor, then the burden of using the processors efficiently is placed on the application programmer. The degree of parallelism, K, of the PAR segments and their total size relative to the sequential segments determine the level of utilization achievable by the multiprocessor.

The most challenging task in achieving high utilization of multiprocessor systems is to develop compilers that can automatically detect parallelism in a user program. The usefulness of automatic detection of parallelism is based on the following reasoning. An application programmer naturally visualizes a program as a set of serially performed operations. However, even though the programmer specifies the operations as a serial list of instructions in some high-level language, many opportunities may exist for executing various groups of instructions in parallel. A simple example is that of successive passes through a loop. If no data dependency is involved between different iterations of the loop, then successive passes can be executed in parallel. On the other hand, if the first pass through the loop generates data that are needed in the second pass, and so

on, then parallel execution is not possible. Data dependencies must be detected by the compiler to determine which operations can be performed in parallel and which cannot. The design of compilers that can detect parallelism is complex. Even after the parallel parts of a program are identified, their subsequent scheduling for execution on a multiprocessor with a limited number of processors is a nontrivial task. Scheduling may be done either by the compiler or at runtime by the operating system. We do not pursue this topic of determining and scheduling tasks that can be executed in parallel. Instead, we turn to the issue of accessing shared variables that are modified by programs running in parallel on different processors of a multiprocessor system.

12.6.1 ACCESSING SHARED VARIABLES

Assume that we have identified two tasks that can run in parallel on a multiprocessor. The tasks are largely independent, but from time to time they access and modify some common, shared variable in the global memory. For example, let a shared variable SUM represent the balance in an account. Moreover, assume that several tasks running on different processors need to update this account. Each task manipulates SUM in the following way: The task reads the current value from SUM, performs an operation that depends on this value, and writes the result back into SUM. It is easy to see how errors can occur if such *read-modify-write* accesses to SUM are performed by tasks T1 and T2 running in parallel on processors P1 and P2. Suppose that both T1 and T2 read the current value from SUM, say 17, and then proceed to modify it locally. T1 adds 5 for a result of 22, and T2 subtracts 7 for a result of 10. They then proceed to write their individual results back into SUM, with T2 writing first followed by T1. The variable SUM now has the value 22, which is wrong. SUM should contain the value 15 $(= 17 + 5 - 7)$, which is the intended result after applying the modifications strictly one after the other, in either order.

To guarantee correct manipulation of the shared variable SUM, each task must have exclusive access to it during the complete read-modify-write sequence. This can be provided by using a global *lock* variable, LOCK, and a machine instruction called Test-and-Set. The variable LOCK has two possible values, 0 or 1. It serves as a guard to ensure that only one task at a time is allowed access to SUM during the time needed to execute the instructions that update the value of this shared variable. Such a sequence of instructions is called a *critical section*. LOCK is manipulated as follows. It is equal to 0 when neither task is in its critical section that operates on SUM. When either task wishes to modify SUM, it first checks the value of LOCK and then sets it to 1, regardless of its original value. If the original value was 0, then the task can safely proceed to work on SUM because no other task is currently doing so. On the other hand, if the original value of LOCK was 1, then the task knows that some other task is operating on SUM. It must wait until that task resets LOCK to 0 before it can proceed. This desired mode of operation on LOCK is made foolproof by the Test-and-Set instruction. As its name implies, this instruction performs the critical steps of testing and setting LOCK in an indivisible sequence of operations executed as a single machine instruction. While this instruction is executing, the memory module involved must not respond to access requests from any other processor.

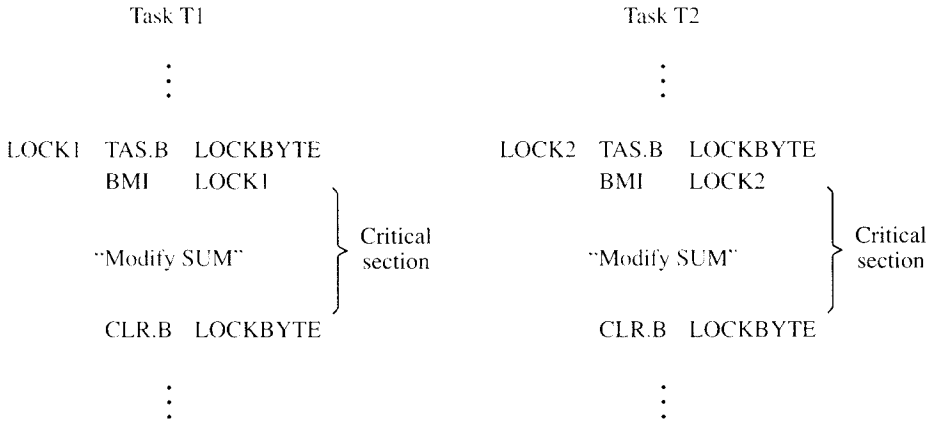


Figure 12.15 Mutually exclusive access to critical sections.

As a specific example, consider the Test-and-Set instruction denoted as TAS in the Motorola 68000 microprocessor. This instruction has one operand that is always a byte. Assume that it is stored in the memory at location LOCKBYTE. Bit b_7, the most significant bit of this operand, serves as the variable LOCK just discussed. The TAS instruction performs the uninterruptible test and set operations on bit b_7. Condition code flag N (Negative) is set to the original value of b_7. Thus, after the execution of TAS is completed, the program can continue into its critical section if N equals 0, but it must wait if N equals 1. Figure 12.15 shows how two tasks, T1 and T2, can manipulate LOCKBYTE to enter critical sections of code in which they update the shared variable SUM. The TAS instruction is followed by a conditional branch instruction. This instruction causes a branch back to TAS if N = 1, resulting in a wait loop that continues to execute TAS on the operand in location LOCKBYTE until it finds b_7 equal to 0. The branch instruction fails if TAS is executed when b_7 is 0, allowing the program to continue into its critical section. When execution of the critical section is completed, LOCKBYTE is cleared. As a result, bit b_7 is reset to 0, allowing any waiting program to proceed into its critical section.

The TAS instruction is an example of a simple machine instruction that can be used to implement a lock. Most computers include an instruction of this type. These instructions may provide additional capabilities, such as incorporating a conditional branch based on the result of the test.

12.6.2 CACHE COHERENCE

Shared data leads to another problem in a multiprocessor machine; the presence of multiple caches means that copies of shared data may reside in several caches. When any processor writes to a shared variable in its own cache, all other caches that contain a copy of that variable will then have the old, incorrect value. They must be informed of the change so that they can either update their copy to the new value or invalidate it.

Cache coherence is defined as the situation in which all cached copies of shared data have the same value at all times.

In Chapter 5 we discussed two basic approaches for performing write operations on data in a cache. The write-through approach changes the data in both the cache and the main memory. The write-back approach changes the data only in the cache; the main memory copy is updated when a dirty data block in the cache has to be replaced. Similar approaches can also be used in a multiprocessor system.

Write-Through Protocol

A write-through protocol can be implemented in two fundamental versions. One version is based on updating the values in other caches, while the second relies on invalidating the copies in other caches.

Let us consider the *write-through with update* protocol first. When a processor writes a new value into its cache, the new value is also written into the memory module that holds the cache block being changed. Since copies of this block may exist in other caches, these copies must be updated to reflect the change caused by the write operation. Conceptually, the simplest way of doing this is to broadcast the written data to all processor modules in the system. As each processor module receives the broadcast data, it updates the contents of the affected cache block if this block is present in its cache (primary or secondary).

The second version of write-through protocol is based on *invalidation* of copies. When a processor writes a new value into its cache, this value is written into the memory module, and all copies in other caches are invalidated. Again, broadcasting can be used to send the invalidation requests throughout the system.

Write-Back Protocol

In the write-back protocol, multiple copies of a cache block may exist if different processors have loaded (read) the block into their caches. If some processor wants to change this block, it must first become an exclusive owner of this block. When the ownership is granted to this processor by the memory module that is the home location of the block, all other copies, including the one in the memory module, are invalidated. Now the owner of the block may change the contents at will without having to take any other action. When another processor wishes to read this block, the data are sent to this processor by the current owner. The data are also sent to the home memory module, which reacquires ownership and updates the block to contain the latest value.

The write-back protocol causes less traffic than the write-through protocol, because a processor is likely to perform several writes to a cache block before this block is needed by another processor.

So far, we have assumed that update and invalidate requests in these protocols are broadcast through the interconnection network. Whether it is practical to implement such broadcasts depends largely on the structure of the interconnection network. The most natural network for supporting broadcasting is the single bus, discussed in Section 12.4.1. In small multiprocessors that use a single bus, cache coherence can be realized using a scheme known as snooping.

Snoopy Caches

In a single-bus system, all transactions between processors and memory modules occur via the bus. In effect, they are broadcast to all units connected to the bus. Suppose that each cache associated with a processor has a controller circuit that observes the transactions on the bus that involve other processors. Suppose also that the write-back protocol just described is used.

Whenever a processor writes to its cache block for the first time, the cache block is marked as dirty, and the write is broadcast on the bus. The memory module and all other caches invalidate their copies. The processor that performed the write is now the owner of the cache block. It can do further writes in the same block without broadcasting them. If another processor issues a read request for the same block, the memory module cannot respond because it does not have a valid copy. But the present owner also sees this request when it appears on the bus, and it must supply the correct value to the requesting processor. The memory module is informed that an owner is supplying the correct value by a broadcast signal from the owner (which includes the data that the owner places on the bus), and the memory updates its value. Finally, the owner marks its copy as clean. Operation now proceeds with multiple caches and the memory module all having the correct value of the block. In the case in which a dirty value must be replaced to make room in the cache for a new block, a write-back operation to the memory module must be performed.

If two processors want to write to the same cache block at the same time, one of the processors will be granted the use of the bus first and will become the owner. As a result, the other processor's copy of the cache block will be invalidated. The second processor can then repeat its write request. This sequential handling of write requests ensures that the two processors can correctly change different words in a given cache block.

The scheme just described is based on the ability of cache controllers to observe the activity on the bus and take appropriate actions. We refer to such schemes as *snoopy-cache* techniques.

For performance reasons, it is important that the snooping function not interfere with the normal operation of a processor and its cache. Such interference would occur if, for each request on the bus, the cache controller had to access the tags of its cache to see if the block in question is present in the cache. In most cases, the answer would be negative. To eliminate unnecessary interference, each cache can be provided with a set of duplicate tags, which maintain the same status information about the blocks in the cache but can be accessed separately by the snooping circuitry.

While the concept of snoopy caches is effective and simple to implement, it is suitable only for single-bus systems. In larger multiprocessors, more complex arrangements must be used.

Directory-Based Schemes

Enforcing cache coherence using a broadcast mechanism for distribution of invalidation or update requests becomes less attractive as the multiprocessor system grows in size. The main reason is that a large amount of unnecessary traffic may be generated by a full broadcast because, in practical applications, copies of a given block are usually present in only a few caches.

A useful alternative is to keep a *directory* of the locations, that is, the caches where copies exist at any given time. One way to implement a directory scheme is to include additional status bits for each block in a particular memory module, which indicate the caches where copies of this block may be found. Then, instead of broadcasting to all caches, the memory module can send individual messages, or a multicast such as an invalidate request in the write-back protocol, to only those caches that have a copy. Of course, the additional bits in the memory modules increase the cost of these modules. Different versions of directory schemes have been proposed and some have been implemented in existing multiprocessor systems.

SCI Standard

A specific approach to cache coherence has been standardized by the Institute of Electrical and Electronics Engineers (IEEE). It is a part of the SCI (Scalable Coherent Interface) standard [9], which defines a multiprocessor backplane that is intended to provide fast signaling, scalable architecture, cache coherence, and simple implementation. The interconnection network uses point-to-point links, and the communication protocol is based on a single-requester single-responder principle. A packet originates at a source node and is addressed to a single target. If a packet sent by the source is accepted by the target, the latter returns a positive acknowledgement packet. If the packet is not accepted, then a negative acknowledgment is returned, which causes a retry.

Cache coherence is achieved using a distributed directory-based protocol. A doubly-linked list is established for each cache block that contains shared data. Each processor node that caches a given block of shared data includes pointers to the previous and to the next nodes that share the block. These pointers are part of the cache-block tag. The head of this doubly-linked list has a pointer to the memory module that holds the block. When a new node accesses the memory module to read this block, the node becomes the new head of the list and the memory directory is updated by replacing the pointer to the previous head with the address of the new head. A write access to the memory can be performed only by the head of the list. If another node wishes to perform a write, it can do so by inserting itself at the head of the list and purging the rest of the entries in the list.

The SCI cache coherence scheme scales well because the memory directory and the processor cache-tag storage requirements do not increase as the size of the linked list increases. The disadvantage of this scheme is that this additional storage presents a costly fixed overhead that is incurred in all cases.

Although the SCI standard does not specify a particular topology for the interconnection network, the ring topology is one of the natural choices. Hewlett-Packard's Exemplar V2600 and Data General's AV25000 multiprocessors use a ring topology and implement the coherence protocol described above.

CC-NUMA Multiprocessors

Cache coherence is an important issue in multiprocessor systems. It has been the topic of extensive research. We have briefly described some key implementation schemes. Many subtle details are beyond the scope of this book.

A multiprocessor may have the cache coherence implemented either in hardware or in software. From the performance point of view, it is advantageous to have hardware-controlled cache coherence. Most of the current NUMA multiprocessors have cache coherence implemented in hardware. They are often referred to as *cache-coherent NUMA* (CC-NUMA) systems.

12.6.3 NEED FOR LOCKING AND CACHE COHERENCE

We should note that the requirement for lock guard controls on access to shared variables is independent of the need for cache coherence controls — both types of controls are needed. Consider a situation in which cache coherence is maintained by using the write-through policy accompanied by cache updating of writes to shared variables. Suppose that the contents of SUM in the example in Section 12.6.1 have been read into the caches of the two processors that execute tasks T1 and T2. If the read operations are part of an update sequence and are not made mutually exclusive by the use of a lock guard control, then the original error can still occur. If task T1 writes its new value last, as before, then SUM will contain the value 22, which is wrong. Cache coherence is maintained throughout this sequence of events. However, incorrect results are obtained because lock guard controls are not used.

12.7 MULTICOMPUTERS

In Section 12.5 we introduced the concept of multicomputers. We now examine the salient features of such systems in more detail.

A multicomputer system is structured as shown in Figure 12.4. Each processing node in the system is a self-contained computer that communicates with other processing nodes by sending messages over the network. Systems of this type are often called *message-passing systems*, in contrast to the shared-memory multiprocessors discussed previously.

In multicomputer systems, the demands on the interconnection network are less stringent than in shared-memory multiprocessor systems. A shared-memory machine must have a fast network with high bandwidth because processor modules frequently access the remote memory modules that constitute the shared memory. A slow network would quickly become a bottleneck, and performance would severely degrade.

In a multicomputer, messages are sent much less frequently, resulting in much less traffic than in the shared-memory systems. Therefore, a simpler and less expensive network can be used. In view of this disparity in the intensity of communication, the terms *tightly coupled* and *loosely coupled* have also been associated with shared-memory and message-passing systems, respectively.

Any network described in Section 12.4 can be used in a multicomputer system. Since the traffic demands are relatively modest, the physical implementation of the interconnection network is likely to be inexpensive. The links in the network often involve bit-serial lines driven by I/O device interfaces. An interface circuit reads a

message from the memory of the source computer using the DMA technique, converts it into a bit-serial format, and transmits it over the network to the destination computer. Source and destination addresses are included in a header of the message for routing purposes. The message is routed to the destination computer where it is written into a memory buffer by the I/O interface of that computer.

In the 1980s, hypercube-based interconnection networks were very popular. Such networks were used in several message-passing multiprocessor systems, typically using bit-serial transmission. Examples of such machines are Intel's iPSC, NCUBE's NCUBE/ten, and Thinking Machines' CM-2. Then in the early 1990s, other topologies gained popularity for both message-passing and shared-memory machines. Thinking Machines' CM-5 is an example of a message-passing machine that uses a fat tree network with a link width of four. Intel's Paragon uses a mesh network with a link width of 16. To facilitate message passing, it is useful to include a special communications unit at each node in the network. For example, the Paragon machine has a message processor that essentially frees the application processor from having to be involved in the details of message handling.

12.7.1 LOCAL AREA NETWORKS

Because the communication demands in a multicomputer system are relatively low, we can consider replacing the specialized interconnection network with some readily available standard network that was developed for more general communication purposes. Many networks exist for interconnecting various types of computing equipment. Networks that span a small geographic area with distances not exceeding a few kilometers are called *local area networks* (LANs). Networks that cover larger areas that involve distances up to thousands of kilometers are referred to as *long-haul networks*, or *wide area networks*.

The most popular LANs use either the bus or the ring topology. The transmission media for either bus or ring LANs can be twisted wire pair, coaxial cable, or optical fiber. Bit-serial transmission is used, and rates range from ten to hundreds of megabits per second. Only one message packet at a time can be successfully transmitted on the single shared path. Source and destination device addresses precede the data field of a packet, and appropriate delimiters indicate the start and end of the packet. In general, packets have variable lengths ranging from tens of bytes to over 1000 bytes.

A protocol that implements distributed access control is needed to ensure orderly transfer of packets between arbitrary pairs of communicating devices. We will sketch the basic ideas involved in two widely used protocols — the Ethernet bus and the token ring. These protocols are specified in detail in IEEE standards [10].

12.7.2 ETHERNET (CSMA/CD) BUS

The Ethernet bus access protocol, also called the Carrier Sense Multiple Access with Collision Detection (CSMA/CD) protocol, is conceptually one of the simplest protocols. Whenever an attached device has a message to transmit, it waits until it senses that the

bus is idle and then begins transmission. The device then monitors the bus for 2τ seconds as it transmits its message, where τ is the end-to-end bus propagation delay. If the device does not observe any distortion of its transmitted signal during the 2τ interval, then it assumes that no other station has started transmission and continues its transmission to completion. On the other hand, if distortion is observed, caused by the beginning of a transmission from some other device, then both devices must stop transmitting. The mutually destructive distortion of the two transmitted signals is called a *collision*, and the time interval 2τ is called the *collision window*.

Messages that have been destroyed by collision must be retransmitted. If the devices involved in the collision attempt to retry immediately, their packets will almost certainly collide again. A basic strategy used to prevent collision of the retries is as follows. Each device independently waits for a random amount of time, then waits until the bus is idle and begins retransmission. If the random waits are a few multiples of 2τ, the probability of repeated collisions is reduced.

12.7.3 TOKEN RING

The token-ring protocol is used for ring networks. A single, appropriately encoded short message, called a *token*, circulates continuously around the ring. The arrival of the token at a ring node represents permission to transmit. If the node has nothing to transmit, it forwards the token to the next node downstream with as little delay as possible. If the node has data ready for transmission, it inhibits propagation of the token. Instead it transmits a packet of information preceded by an appropriately encoded header flag. As the packet is transmitted around the ring, its contents are read and copied as it travels past the destination node. The packet continues to travel around the ring until it reaches the source node, where it is discarded. When the source node completes transmitting a packet, it releases the token, which again starts to circulate around the ring. The packet size on a token ring is variable and is limited only by the amount of buffer memory available in each node because the destination node must be able to store complete packets.

The main reason for considering the standard LANs in the context of multicomputer systems is not because they can be used in self-contained systems that we have been discussing, but because they can be used in conjunction with standard workstations to conveniently form a multicomputer system.

12.7.4 NETWORK OF WORKSTATIONS

Today, most commercial, educational, and government organizations have a collection of workstations to meet their computing needs. These workstations are usually connected to a LAN that allows access to fileservers, printers, and specialized computing resources. (See Figure 12.16.)

Although each workstation is normally used as a separate computer, many workstations can be viewed as a multicomputer system. All that is needed is the software to allow parallel processing. Of course, some significant differences exist between such a system and a commercial message-passing multiprocessor machine. In particular,

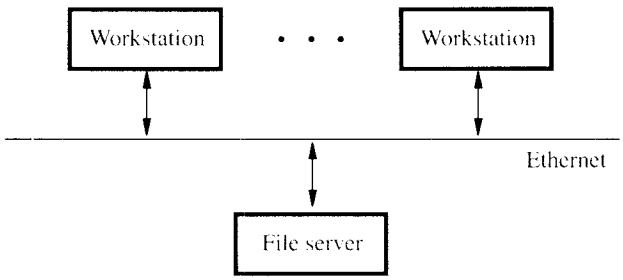


Figure 12.16 A typical network of workstations.

communication over the LAN is slower, largely because the operating system must intervene when messages have to be exchanged between programs running on different computers. This means that a network of workstations does not perform as well as a self-contained system with a specialized interconnection network. But the great advantage is that the network of workstations is usually readily available. It is certainly useful to be able to run very large applications on such systems when the workstations are not used for their normal purposes, which is typically the case at night.

12.8 PROGRAMMER'S VIEW OF SHARED MEMORY AND MESSAGE PASSING

In previous sections, we considered the hardware implications of multiprocessor systems that feature shared-memory and message-passing paradigms. Now we briefly examine how these paradigms affect the user, namely, the programmer who is implementing a parallel application. We consider a small example that involves only two processors. This keeps the discussion simple yet allows us to elaborate the key ideas.

Assume we want to compute the dot product of two N-element vectors. A sequential program for this task is outlined in Figure 12.17. It is suitable for execution on a single processor. The program is mostly self-explanatory. The **read** statements load the values of the two vectors from a disk (or some other I/O device) into the main memory. This task is done by the operating system. Let us attempt to parallelize this program to run on two processors. Evidently, the potential for parallelization lies in the loop that computes the dot product by generating the product of a pair of elements and adding the result to the previously accumulated partial dot product.

12.8.1 SHARED MEMORY CASE

Our first attempt to write a program for two processors is shown in Figure 12.18. As the program starts executing on one processor, it loads the vectors into memory and initializes the *dot_product* variable to 0. We achieve parallelism by having a second

```
integer array a[1..N], b[1..N]
integer dot_product
.
.
read a[1..N] from vector_a
read b[1..N] from vector_b
dot_product := 0
do_dot (a, b)
print dot_product
.
.
do_dot (integer array x[1..N], integer array y[1..N])
        for k:= 1 to N
                dot_product := dot_product + x[k] * y[k]
        end
end
```

Figure 12.17 A sequential program to compute the dot product.

```
shared integer array a[1..N], b[1..N]
shared integer dot_product
shared lock dot_product_lock
shared barrier done
.
.
read a[1..N] from vector_a
read b[1..N] from vector_b
dot_product := 0
create_thread (do_dot, a, b)
do_dot (a, b)
print dot_product
.
.
do_dot (integer array x[1..N], integer array y[1..N])
        private integer id
        id := mypid()
        for k:= (id*N/2) + 1 to (id + 1)*N/2
                lock (dot_product_lock)
                        dot_product := dot_product + x[k] * y[k]
                unlock (dot_product_lock)
        end
        barrier (done)
end
```

Figure 12.18 A first attempt at a program to compute the dot product on two processors in a shared memory machine.

processor perform half of the computations needed to obtain the dot product. This is done by creating a separate thread to be executed on the second processor.

A thread is an independent path of execution within a program. Actually, the term *thread* is used to refer to a thread of control, where multiple threads execute portions of the program and can run in parallel as if they were separate programs. Thus, two or more threads can be running on different processors, executing either the same or different code. The key point is that all threads are part of a single program and run in the same address space. We should note that in the commonly used uniprocessor environment, each program has a single thread of control.

In the program in Figure 12.18, a new thread is created by the *create_thread* statement. This thread will execute the *do_dot* routine and terminate. The operating system will assign the identification number of 1 to the new thread. The first processor continues by executing the *do_dot (a,b)* statement as thread 0. The statement *id := mypid()* sets the variable *id* to the assigned identification number of the thread. Using the *id* value in the **for** loop allows simple specification of which halves of the vectors *a* and *b* should be handled by a particular thread.

Changing the accumulated value of the *dot_product* variable is the critical section in the *do_dot* routine; hence, each thread must have exclusive access to this variable. This is achieved by the locking mechanism, as discussed in Section 12.6.1. Thread 0 does not proceed past the barrier statement in the *do_dot* routine until the other thread has reached the same synchronization point. This ensures that both threads have completed their updates of the *dot_product* variable before thread 0 is allowed to print the final result. The *barrier* concept can be realized in different ways. A simple approach is to use a shared variable, such as *done* in Figure 12.18. This variable is initialized to the number of threads (two in our example) and then decremented as each thread arrives at the barrier.

The program in Figure 12.18 has one major flaw. The locking arrangement used does not allow the expected parallelism to be achieved because both threads continuously write the same shared variable, *dot_product*. Thus, the potentially parallel part of the required computation will in fact be done serially.

To achieve the desired parallelism, we can modify the program as shown in Figure 12.19. Instead of using the shared variable, *dot_product*, in the **for** loop, a private variable, *local_dot_product*, is introduced to accumulate the partial dot product as it is being computed by each thread. Thus, only upon completion of the loop is it necessary to enter a critical section where each thread updates the shared variable, *dot_product*. This modification allows both threads to execute the **for** loop in parallel.

This example can be easily extended to a larger number of processors. All that needs to be done is to create more threads. The loop bound expressions in the **for** loop will determine the range of elements that each thread uses in the computation based on the value of the assigned *id*.

The effectiveness of the program in Figure 12.19 depends on the size of the data vectors. The larger the vectors, the more effective this approach is. For small vectors, the overhead of creating threads and providing synchronization outweighs any benefit that parallelism may provide.

```
shared integer array a[1..N], b[1..N]
shared integer dot_product
shared lock dot_product_lock
shared barrier done
.
.
read a[1..N] from vector_a
read b[1..N] from vector_b
dot_product := 0
create_thread (do_dot, a, b)
do_dot (a, b)
print dot_product
.
.
do_dot (integer array x[1..N], integer array y[1..N])
        private integer local_dot_product
        private integer id
        id := mypid()
        local_dot_product := 0
        for k:= (id*N/2) + 1 to (id + 1)*N/2
                local_dot_product := local_dot_product + x[k] * y[k]
        end
        lock (dot_product_lock)
                dot_product := dot_product + local_dot_product
        unlock (dot_product_lock)
        barrier (done)
end
```

Figure 12.19 An efficient program to compute the dot product on two processors in a shared memory machine.

12.8.2 MESSAGE-PASSING CASE

In this case the memory is distributed, and each processor can access directly only its own memory. The desired program will run on two processors and the arrays will have to be explicitly divided into halves, with each half being stored in the memory of one processor. Each copy of the program will have access only to its portion of the data. Applications of this type are called *Single Program Multiple Data* (SPMD). The reader should note the difference between this type of application and the SIMD type introduced in Section 12.1.1. In the SIMD type, all processors execute the same instruction at any given time.

A possible program is given in Figure 12.20. The vector data must first be loaded into the private memories of the two processors. The program that is assigned the *id* value of 0 reads the first half of vector *a* from the disk, with the help of the operating

```
integer array a[1..N/2], b[1..N/2], temparray[1..N/2]
integer dot_product
integer id
integer temp
.
.
id := mypid()
if (id = 0) then
            read a[1..N/2] from vector_a
            read temparray[1..N/2] from vector_a
            send (temparray[1..N/2], 1)
            read b[1..N/2] from vector_b
            read temparray[1..N/2] from vector_b
            send (temparray[1..N/2], 1)
      else  receive (a[1..N/2], 0)
            receive (b[1..N/2], 0)
end
dot_product := 0
do_dot (a, b)
if (id = 1)  send (dot_product, 0)
      else  receive (temp, 1)
            dot_product := dot_product + temp
            print dot_product
end
.
.
do_dot (integer array x[1..N/2], integer array y[1..N/2])
      for k:= 1 to N/2
            dot_product := dot_product + x[k] * y[k]
      end
end
```

Figure 12.20 A message-passing program to compute the dot product on two processors.

system, and it stores the data in its memory under this name. It then reads the remaining second half of vector *a* and places the data in a memory buffer called *temparray*. Next, it sends a message containing the data from this buffer to the processor that executes the program that is assigned the *id* value of 1. The same operations are then repeated for the data that constitute vector *b*. The program with the *id* value of 1 receives the second halves of vectors *a* and *b* and stores them in its memory under the same names.

The *do_dot* routine now simply computes the dot product for the N/2 elements. Note that the loop bounds are the same for both processors because each uses the data stored in its own memory. The message-passing feature is also illustrated by the action taken when the processors complete execution of the *do_dot* routine: The program that has the *id* value of 0 will compute and print the final dot product. It will do so when it

receives the message with the value of the partial dot product that was computed and sent by program 1. This value is received in a temporary buffer called *temp*.

Again, it is easy to see how this example could be extended to many processors. The vectors would have to be partitioned into portions that would be assigned to each processor for computation. One of the processors, for example, the one that executes the program with $id = 0$, would be designated to compute the final result using the data received in messages from other processors.

The overhead of establishing parallel execution on multiple processors consists of the time needed to load the copies of the program into different processors, the time used to set up the partitioned arrays in the memories associated with different processors, and the time needed to send other messages among processors. Performance benefits depend on the size of the vectors and the number of processors used.

Shared-memory and message-passing paradigms have certain strengths and weaknesses. The shared-memory environment is more natural to use because it is an extension of the uniprocessor programming model. Hence, it is easier to write parallel programs that are reasonably efficient. Since the memory access latency may be high if data reside in remote memory modules, it is important to minimize the number of write accesses to global variables. The amount of traffic in the network may be large, causing the network to become a bottleneck. Synchronization of processes is the responsibility of the programmer and influences the performance of an application significantly.

Message passing gives a less natural programming environment because of multiple address spaces in private memories. The time overhead of message passing is very significant; hence, the programmer must try to structure programs to minimize its effect. Since messages are relatively infrequent, the interconnection network is not likely to be a problem. Synchronization is implicit in the messages passed between processes. Perhaps the biggest advantage of message passing is that it can be supported by less expensive and more commonly available hardware.

12.9 PERFORMANCE CONSIDERATIONS

This chapter has concentrated on the design of systems that use multiple processors to reduce the time needed to run a large application. The most important performance measure is the speedup achieved on a multiprocessor system in comparison with the time it would take to run the same application on a single processor. The *speedup* is defined as

$$S_P = \frac{T_1}{T_P}$$

where T_1 and T_P are the times needed if one or P processors are used, respectively. Figure 12.21 shows three types of speedup that may occur as a function of the number of processors in the system. Intuitively, we would expect that, as the number of processors is increased, the time needed to run an application that is parallelizable should decrease proportionately. This would give a linear speedup, where $S = P$, which is the goal in scalable systems. Unfortunately, this goal is not easy to achieve.

As the previous section shows, it is not possible to parallelize all parts of an application program. The sequential parts will take the same amount of time regardless

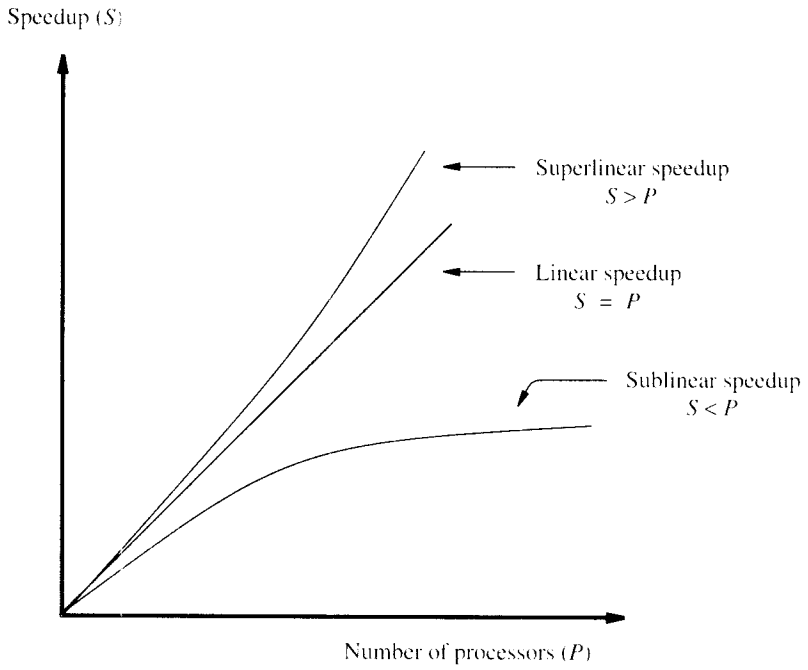


Figure 12.21 Speedup curves in multiprocessor systems.

of the number of processors used. It is the relative proportion of sequential parts that limits the achievable speedup.

Another reason why linear speedup is difficult to achieve is the overhead caused by initialization, synchronization, communication, cache coherence control, and load imbalance. Overhead tends to increase with the size of the system. We have encountered examples of such overhead in previous sections, except for load imbalance. It is usually necessary to wait for the last processor to complete a parallel task before proceeding with the next set of tasks. Hence, when a parallel task is spread over a number of processors, it is most efficient if all processors reach a given synchronization point at about the same time, in which case the load is balanced.

In practical systems the speedup achievable with most applications is sublinear, and a point is reached where adding processors no longer improves performance. Curiously, there exist some applications for which even superlinear speedup is possible, but these are not common. We give an example of such an application in the next subsection.

12.9.1 AMDAHL'S LAW

Let us consider the improvement in performance from a quantitative point of view. An enhancement in a computer system inevitably improves some part of the system but not the entire system. The improved performance depends on the impact of the enhanced part. This reasoning was formalized by Gene Amdahl in a well known "law" [11].

It can be stated as

$$S_{\text{new}} = \frac{\textit{Old time}}{\textit{New time}} = \frac{1}{1 - f_{\text{enhanced}} + f_{\text{enhanced}}/S_{\text{enhanced}}}$$

where

- S_{new} is the speedup in the new system, which includes the enhancement.
- S_{enhanced} is the speedup achievable if only the enhanced part of the system is used.
- f_{enhanced} is the fraction of the computation time in the old system that can be improved with the enhancement made.

In terms of a multiprocessor system, this law can be restated as follows. Let f be the fraction of a computation (in terms of time) that is parallelizable, P be the number of processors in the system, and S_P be the speedup achievable in comparison with sequential execution. Then we have

$$S_P = \frac{1}{1 - f + f/P} = \frac{P}{P - f(P - 1)}$$

This formula assumes that the parallelizable part is performed by all processors using perfect load balance.

Suppose that a given application is run on a 64-processor machine and that 70 percent of the application is parallelizable. Then the expected improvement is

$$S_{64} = \frac{64}{64 - 0.7 \times 63} = 3.22$$

If the same application is run on a 16-processor machine, the expected speedup would be 2.91. This indicates that the speedup is much less than the number of processors in the machine. Moreover, the difference in speedup achieved by increasing from 16 to 64 processors is minimal. Clearly, it makes little sense to use large multiprocessors for applications that have significant sequential (nonparallelizable) parts. For good speedup, the sequential parts must be very short. For an application with $f = 0.95$, the speedup in the preceding machines would be 15.42 and 9.14, respectively. Amdahl's law, in fact, states that linear speedup cannot be achieved because almost all applications have some sections that cannot be parallelized.

This discussion assumes that each processor performs an equal amount of parallel computation. Such equal load balancing may not necessarily occur. If it is necessary to wait for the slowest processor to complete its parallel assignment before continuing with the next step, then the results will be worse than predicted by the preceding formula. However, there exist applications where the opposite may occur, namely, where the tasks performed by all processors may be terminated as soon as one processor completes its task. For example, such unusual behavior occurs in applications based on a technique known as simulated annealing. To illustrate this technique, suppose that in the design of a VLSI chip, it is desired to place the logic gates such that the total length of wires in the resulting circuit is minimized. This requires trying a large number of different placements, which can be done by assigning the best placement known at a given time to all processors as a starting point for the next iteration. Then each processor can use a different randomized approach to change the positions of the gates in search for a better

placement. As soon as one processor finds a placement that is superior to the starting placement by some predetermined amount, this processor's solution can be used as the new starting point for all the processors, without waiting for the other processors to also find acceptable solutions. An application of this type may exhibit superlinear speedup, because if it is performed by a single processor, this processor may spend a lot of time investigating unpromising possibilities before it reaches a good one.

12.9.2 PERFORMANCE INDICATORS

From a user's point of view, the most important characteristics of a computer system are its cost, ease of use, reliability, and performance. Several indicators of performance are used to depict the processing capability of computers. The discussion of this issue in Section 8.8 applies equally to multiprocessor systems.

The raw power of a processor can be indicated in terms of the number of operations it can perform in one second. Two popular measures are MIPS, the number of millions of instructions executed per second, and MFLOPS (pronounced megaFLOPS), the number of millions of floating-point operations performed per second. When a manufacturer gives the MIPS and MFLOPS numbers for a given processor, these numbers indicate the processor's maximum capability. This maximum is not always achievable in practical applications. In a multiprocessor system, the total MIPS and MFLOPS are simply the sums of the values for all the processors.

Another common performance indicator is the communications capability of the interconnection network, usually given as the total bandwidth in bytes per second. This assumes an optimal situation in which sufficient data are available for transfer to keep the largest possible number of network links busy, thus maximizing the amount of data that can be transferred at one time.

While indicators such as MIPS, MFLOPS, and network bandwidth give a useful impression of what the system is capable of doing, they are not a measure of the actual performance we expect to observe when application programs are executed. Practical applications can use only a fraction of the total resources available at any given time. This fraction varies from one system to another and from one application to another. A proper comparison of two different systems is possible only if a desired set of applications is run on both systems and their performance is observed. To facilitate such comparisons, a number of *benchmark* programs have been developed. These programs are indicative of the behavior found in a variety of common applications. Comparing different systems based on benchmark programs has become widely accepted.

12.10 CONCLUDING REMARKS

Multiprocessors provide a way to realize supercomputing capability at a reasonable cost. They are most cost-effective in the range of tens to hundreds of processors. Very large systems comprising thousands of processors are difficult to use fully, and their cost curtails the market demand significantly.

A particularly cost-effective possibility is to implement a multicomputer system using workstations interconnected by a local area network. This possibility will become even more attractive as local area network speeds increase.

Successful use of multiprocessors depends heavily on the availability of system software that makes good use of the available resources. An application program will not show good performance if the locality and parallelism inherent in the application are not properly exploited. The compiler must detect the opportunities for parallel execution. The operating system must schedule the execution to make good use of locality, by assigning tasks that involve a large amount of interaction to processors that are close to each other. The application programmer may provide useful hints in this respect, but it is best if the system software can do this on its own.

This chapter provides an overview of the most important aspects of multiprocessor and multicomputer systems. Many details should be studied to understand fully the capabilities of these systems and the design issues involved. For detailed study, the reader should consult books that focus on this subject [12–16].

PROBLEMS

12.1 Write a program loop whose instructions can be broadcast from the control processor in Figure 12.1 that will enable an array processor to iteratively compute temperatures in a plane, as discussed in Section 12.2. In addition to instructions that shift the network register contents between adjacent processing elements (PEs), assume that there are two-operand instructions for moves between PE registers and local memory and for arithmetic operations. Assume also that each PE stores the current estimate of its grid point temperature in a local memory location named CURRENT and that a few registers, R0, R1, and so on, are available for processing. Each boundary PE maintains a fixed boundary temperature value in its network register and does not execute the broadcast program. A small value stored in location EPSILON in each PE is used to determine when the local temperature has reached the required level of accuracy. At the end of each iteration of the loop, each PE must set its status bit, STATUS, to 1 if its new temperature satisfies the following condition:

$$|\text{New temperature} - [\text{CURRENT}]| < [\text{EPSILON}]$$

Otherwise, STATUS is set to 0.

12.2 Assume that a bus transfer takes T seconds and memory access time is $4T$ seconds. A read request over a conventional bus then requires $6T$ seconds to complete. How many conventional buses are needed to equal or exceed the bandwidth of a split-transaction bus that operates with the same time delays? Consider only read requests, ignore memory conflicts, and assume that all memory modules are connected to all buses in the multiple bus case. Does your answer increase or decrease if memory access time increases?

12.3 In a bus-based multiprocessor, the system bus can become a bottleneck if it does not support a high enough transfer rate. Suppose that a split-transaction bus is designed to

be four times as wide as the word length of the processors used in the system. Will this increase the effective transfer rate to four times the rate of a similar bus that is only as wide as the processor word length? Explain your answer.

12.4 Assume that the cost of a 2×2 switch in a shuffle network is twice the cost of a crosspoint in a crossbar switch. There are n^2 crosspoints in an $n \times n$ crossbar switch. As n increases, the crossbar becomes more costly than the shuffle network. What is the smallest value of n for which crossbar cost is five times more costly than the shuffle network?

12.5 Shuffle networks can be built from 4×4 and 8×8 switches, for example, instead of from 2×2 switches. Draw a 16×16 ($n = 16$) shuffle network built from 4×4 switches. If the cost of a 4×4 switch is four times the cost of a 2×2 switch, compare the cost of shuffle networks built from 4×4 switches with those built from 2×2 switches for n values in the sequence $4, 4^2, 4^3$, and so on. Qualitatively compare the blocking probability of these two different ways of building shuffle networks.

12.6 Suppose that each procedure of a PAR segment (see Figure 12.14) requires 1 unit of time to execute. A program consists of three sequential segments. Each segment requires k time units and must be executed on a single processor. The three sequential segments are separated by two PAR segments, each of which consists of k procedures that can be executed on independent processors. Derive an expression for speedup for this program when it is run on a multiprocessor with n processors. Assume $n \leq k$. What is the limiting value of the speedup when k is large and $n = k$? What does this result tell you about the effect of sequential segments in programs that have some segments with substantial parallelism?

12.7 The shortest distance a message travels in an n-dimensional hypercube is 1 hop, and the longest distance a message needs to travel is n hops. Assuming that all possible source/destination pairs are equally likely, is the average distance a message needs to travel larger or smaller than $(1 + n)/2$? Justify your answer.

12.8 A task that "busy-waits" on a lock variable by using a Test-and-Set instruction in a two-instruction loop, as in Figure 12.15, wastes bus cycles that could otherwise be used for computation. Suggest a way around this problem that involves a centralized queue of waiting tasks that is maintained by the operating system. Assume that the operating system can be called by a user task and that the operating system chooses which task is to be executed on a processor from among those ready for execution.

12.9 What are the arguments for and against invalidation and updating as strategies for maintaining cache coherence?

12.10 Section 12.6.3 argues that cache coherence controls cannot replace the need for lock variables. Can the use of lock variables replace the need for explicit cache coherence controls?

12.11 Estimate the improvement in performance that can be achieved if the program in Figure 12.19 is used rather than the program in Figure 12.18. Make some appropriate assumptions about the amount of time it takes to perform each step in the program.

12.12 Modify the program in Figure 12.19 to make it suitable for execution in a four-processor machine.

12.13 Modify the program in Figure 12.20 to make it suitable for execution in a four-processor system.

12.14 For small vectors, the approach in Figure 12.19 will be worse than if the dot product is computed using a single processor. Estimate the minimum size of the vectors for which this approach leads to better performance. Make some appropriate assumptions about the amount of time it takes to perform each step in the program.

12.15 Repeat Problem 12.14 for the approach in Figure 12.20.

12.16 Shared-memory multiprocessors and message-passing multicomputers are architectures that support simultaneous execution of tasks that interact with each other. Which of these two architectures can emulate the action of the other more easily? Briefly justify your answer.

12.17 The Ethernet bus LAN protocol is really only suitable when message transmission time is significantly larger than 2τ, where τ is the end-to-end bus propagation delay. Consider the case in which transmission time is less than τ. Is it possible for a destination station to correctly receive an undistorted message, even though the source station observes a collision inside the 2τ collision window period? If not, justify your answer. If you think it is possible, give the relative locations of the source, destination, and interfering stations on the bus and describe the relevant event times.

12.18 A *mailbox memory* is a RAM memory with the following feature. A full/empty bit, F/E, is associated with each memory word location. The instruction

```
PUT   R0,BOXLOC,WAITSEND
```

is executed indivisibly as follows. The F/E bit associated with mailbox memory location BOXLOC is tested. If it is 0, denoting empty, then the contents of register R0 are written into BOXLOC, F/E is set to 1, denoting full, and execution continues with the next sequential instruction. Otherwise (that is, for F/E = 1), no operations are performed and execution control is passed to the instruction at location WAITSEND in program memory.

(*a*) Give an appropriate definition for the instruction

```
GET   R0,BOXLOC,WAITREC
```

that is complementary to the PUT instruction.

(*b*) Suppose two tasks, T_1 and T_2, running on different processors in a multiprocessor system, pass a stream of one-word messages from T_1 to T_2 using PUT and GET instructions on a shared mailbox memory unit. Write program segments for T_1 and T_2 in assembly-language style that accomplish the same thing on a shared-memory multiprocessor system that does not have a mailbox memory unit but does have a TAS instruction as described in Section 12.6.1.

REFERENCES

1. M.J. Flynn, "Very High-Speed Computing Systems," *Proceedings of the IEEE*, vol. 54, December 1966, pp. 1901–1909.
2. D.L. Slotnick, "The Fastest Computer," *Scientific American*, vol. 224, February 1971, pp. 76–88.
3. D. Lenoski, et al., "The Stanford DASH Multiprocessor," *Computer*, vol. 25, March 1992, pp. 63–79.
4. J. Kuskin, et al., "The Stanford FLASH Multiprocessor," *Proceedings of the 21st Annual International Symposium on Computer Architecture*, Chicago, April 1994, pp. 302–313.
5. A. Agarwal, et al., "The MIT Alewife Machine: Architecture and Performance," *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995, pp. 2–13.
6. Z.G. Vranesic, M. Stumm, D.M. Lewis, and R. White, "Hector: A Hierarchically Structured Shared-Memory Multiprocessor," *Computer*, vol. 24, January 1991, pp. 72–79.
7. R. Grindley, et al., "The NUMAchine Multiprocessor," *Proceedings of the 2000 International Conference on Parallel Processing*, Toronto, Ont., August 2000, pp. 487–496.
8. W.J. Dally and P. Song, "Design of a Self-Timed Multicomputer Communication Controller," *Proceedings of the 1987 International Conference on Computer Design*, October 1987, pp. 230–234.
9. D. Gustavson, "The Scalable Coherent Interface and Related Standards Projects," *IEEE Micro*, vol. 12, January 1992, pp. 10–22.
10. *IEEE Local Area Standard 802*, IEEE, 1985.
11. G.M. Amdahl, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities," *Proceedings of AFIPS Spring Joint Computer Conference*, Atlantic City, NJ, April 1967, pp. 483–485.
12. D. Culler, J.P. Singh, and A. Gupta, *Parallel Computer Architecture — A Hardware/Software Approach*, Morgan Kaufmann, San Francisco, CA, 1998.
13. G.S. Almasi and A. Gottlieb, *Highly Parallel Computing*, 2nd ed., Benjamin-Cummings, Redwood City, CA, 1994.
14. K. Hwang, *Advanced Computer Architecture*, McGraw-Hill, New York, 1993.
15. H.S. Stone, *High-Performance Computer Architecture*, 3rd ed., Addison-Wesley, Reading, MA, 1993.
16. D. Tabak, *Multiprocessors*, Prentice-Hall, Englewood Cliffs, NJ, 1990.