1.3 TYPES OF PARALLEL COMPUTERS

Having convinced ourselves that there is potential for speedup with the use of multiple processors or computers, let us explore how a multiprocessor or multicomputer could be constructed. A parallel computer, as we have mentioned, is either a single computer with multiple internal processors or multiple computers interconnected to form a coherent

high-performance computing platform. In this section, we shall look at specially designed parallel computers, and later in the chapter we will look at using an off-the-shelf "commodity" computer configured as a cluster. The term *parallel computer* is usually reserved for specially designed components. There are two basic types of parallel computer:

1. Shared memory multiprocessor
2. Distributed-memory multicomputer.

1.3.1 Shared Memory Multiprocessor System

A conventional computer consists of a processor executing a program stored in a (main) memory, as shown in Figure 1.5. Each main memory location in the memory is located by a number called its *address*. Addresses start at 0 and extend to $2^b - 1$ when there are b bits (binary digits) in the address.

A natural way to extend the single-processor model is to have multiple processors connected to multiple memory modules, such that each processor can access any memory module in a so–called *shared memory* configuration, as shown in Figure 1.6. The connection between the processors and memory is through some form of *interconnection network.* A shared memory multiprocessor system employs a *single address space,* which means that each location in the whole main memory system has a unique address that is used by each processor to access the location. Although not shown in these "models," real systems have high-speed cache memory, which we shall discuss later.

Programming a shared memory multiprocessor involves having executable code stored in the shared memory for each processor to execute. The data for each program will also be stored in the shared memory, and thus each program could access all the data if

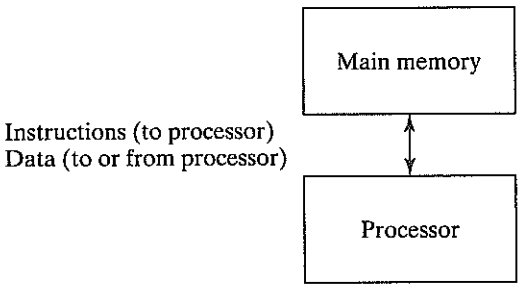

Figure 1.5 Conventional computer having a single processor and memory.

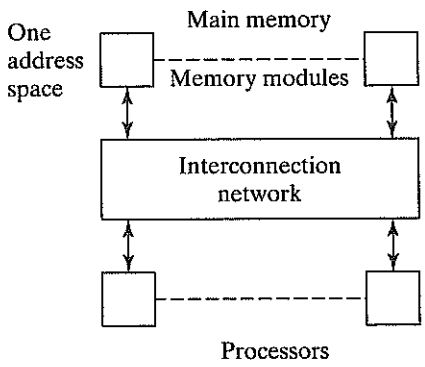

Figure 1.6 Traditional shared memory multiprocessor model.

needed. A programmer can create the executable code and shared data for the processors in different ways, but the final result is to have each processor execute its own program or code sequences from the shared memory. (Typically, all processors execute the same program.)

One way for the programmer to produce the executable code for each processor is to use a high-level parallel programming language that has special parallel programming constructs and statements for declaring shared variables and parallel code sections. The compiler is responsible for producing the final executable code from the programmer's specification in the program. However, a completely new parallel programming language would not be popular with programmers. More likely when using a compiler to generate parallel code from the programmer's "source code," a regular sequential programming language would be used with preprocessor directives to specify the parallelism. An example of this approach is OpenMP (Chandra et al., 2001), an industry-standard set of compiler directives and constructs added to C/C++ and Fortran. Alternatively, so-called *threads* can be used that contain regular high-level language code sequences for individual processors. These code sequences can then access shared locations. Another way that has been explored over the years, and is still finding interest, is to use a regular sequential programming language and modify the syntax to specify parallelism. A recent example of this approach is UPC (Unified Parallel C) (see http://upc.gwu.edu). More details on exactly how to program shared memory systems using threads and other ways are given in Chapter 8.

From a programmer's viewpoint, the shared memory multiprocessor is attractive because of the convenience of sharing data. Small (two-processor and four-processor) shared memory multiprocessor systems based upon a bus interconnection structure-as illustrated in Figure 1.7 are common; for example dual-Pentium® and quad-Pentium systems. Two-processor shared memory systems are particularly cost-effective. However, it is very difficult to implement the hardware to achieve fast access to all the shared memory by all the processors with a large number of processors. Hence, most large shared memory systems have some form of hierarchical or distributed memory structure. Then, processors can physically access nearby memory locations much faster than more distant memory locations. The term *nonuniform memory access* (NUMA) is used in these cases, as opposed to *uniform memory access* (UMA).

Conventional single processors have fast cache memory to hold copies of recently referenced memory locations, thus reducing the need to access the main memory on every memory reference. Often, there are two levels of cache memory between the processor and the main memory. Cache memory is carried over into shared memory multiprocessors by providing each processor with its own local cache memory. Fast local cache memory with each processor can somewhat alleviate the problem of different access times to different main memories in larger systems, but making sure that copies of the same data in different

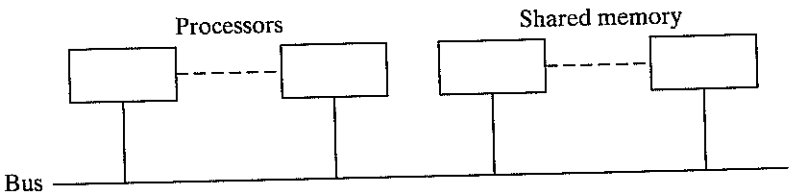

Figure 1.7 Simplistic view of a small shared memory multiprocessor.

caches are identical becomes a complex issue that must be addressed. One processor writing to a cached data item often requires all the other copies of the cached item in the system to be made invalid. Such matters are briefly covered in Chapter 8.

1.3.2 Message-Passing Multicomputer

An alternative form of multiprocessor to a shared memory multiprocessor can be created by connecting complete computers through an interconnection network, as shown in Figure 1.8. Each computer consists of a processor and local memory but this memory is not accessible by other processors. The interconnection network provides for processors to send messages to other processors. The messages carry data from one processor to another as dictated by the program. Such multiprocessor systems are usually called *message-passing multiprocessors*, or simply *multicomputers*, especially if they consist of self-contained computers that could operate separately.

Programming a message-passing multicomputer still involves dividing the problem into parts that are intended to be executed simultaneously to solve the problem. Programming could use a parallel or extended sequential language, but a common approach is to use message-passing library routines that are inserted into a conventional sequential program for message passing. Often, we talk in terms of *processes*. A problem is divided into a number of concurrent processes that may be executed on a different computer. If there were six processes and six computers, we might have one process executed on each computer. If there were more processes than computers, more than one process would be executed on one computer, in a time-shared fashion. Processes communicate by sending messages; this will be the only way to distribute data and results between processes.

The message-passing multicomputer will physically *scale* more easily than a shared memory multiprocessor. That is, it can more easily be made larger. There have been examples of specially designed message-passing processors. Message-passing systems can also employ general-purpose microprocessors.

Networks for Multicomputers. The purpose of the interconnection network shown in Figure 1.8 is to provide a physical path for messages sent from one computer to another computer. Key issues in network design are the *bandwidth*, *latency*, and *cost*. Ease of construction is also important. The *bandwidth* is the number of bits that can be transmitted in unit time, given as bits/sec. The *network latency* is the time to make a message transfer through the network. The *communication latency* is the total time to send the

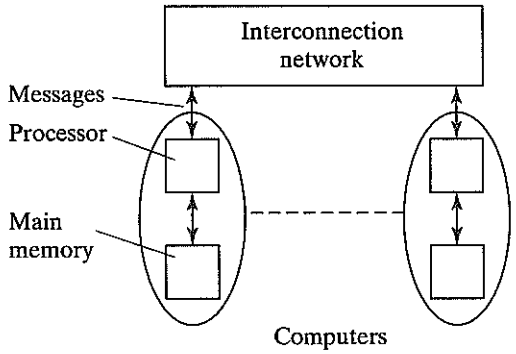

Figure 1.8 Message-passing multiprocessor model (multicomputer).

message, including the software overhead and interface delays. *Message latency,* or *startup time*, is the time to send a zero-length message, which is essentially the software and hardware overhead in sending a message (finding the route, packing, unpacking, etc.) onto which must be added the actual time to send the data along the interconnection path.

The number of physical links in a path between two nodes is an important consideration because it will be a major factor in determining the delay for a message. The *diameter* is the minimum number of links between the two farthest nodes (computers) in the network. Only the shortest routes are considered. How efficiently a parallel problem can be solved using a multicomputer with a specific network is extremely important. The diameter of the network gives the maximum distance that a single message must travel and can be used to find the communication lower bound of some parallel algorithms.

The *bisection width* of a network is the minimum number of links (or sometimes wires) that must be cut to divide the network into two equal parts. The *bisection bandwidth* is the collective bandwidth over these links, that is, the maximum number of bits that can be transmitted from one part of the divided network to the other part in unit time. These factor can also be important in evaluating parallel algorithms. Parallel algorithms usually require numbers to be moved about the network. To move numbers across the network from one side to the other we must use the links between the two halves, and the bisection width gives us the number of links available.

There are several ways one could interconnect computers to form a multicomputer system. For a very small system, one might consider connecting every computer to every other computer with links. With c computers, there are $c(c-1)/2$ links in all. Such exhaustive interconnections have application only for a very small system. For example, a set of four computers could reasonably be exhaustively interconnected. However, as the size increases, the number of interconnections clearly becomes impractical for economic and engineering reasons. Then we need to look at networks with restricted interconnection and switched interconnections.

There are two networks with restricted direct interconnections that have seen wide use — the *mesh* network and the *hypercube* network. Not only are these important as interconnection networks, the concepts also appear in the formation of parallel algorithms.

Mesh. A two-dimensional *mesh* can be created by having each node in a two-dimensional array connect to its four nearest neighbors, as shown in Figure 1.9. The diameter of a $\sqrt{p} \times \sqrt{p}$ mesh is $2(\sqrt{p}-1)$, since to reach one corner from the opposite corner requires a path to made across $(\sqrt{p}-1)$ nodes and down $(\sqrt{p}-1)$ nodes. The free ends of a mesh might circulate back to the opposite sides. Then the network is called a *torus*.

The mesh and torus networks are popular because of their ease of layout and expandability. If necessary, the network can be *folded*; that is, rows are interleaved and columns are interleaved so that the wraparound connections simply turn back through the network rather than stretch from one edge to the opposite edge. Three-dimensional meshes can be formed where each node connects to two nodes in the x-plane, the y-plane, and the z-plane. Meshes are particularly convenient for many scientific and engineering problems in which solution points are arranged in two-dimensional or three-dimensional arrays.

There have been several examples of message-passing multicomputer systems using two-dimensional or three-dimensional mesh networks, including the Intel Touchstone Delta computer (delivered in 1991, designed with a two-dimensional mesh), and the J-machine, a

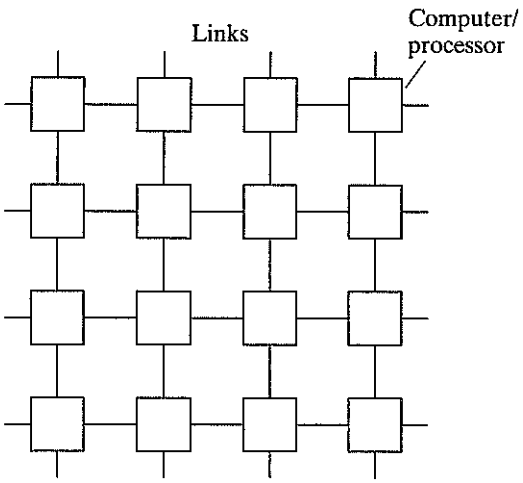

Figure 1.9 Two-dimensional array (mesh).

research prototype constructed at MIT in 1991 with a three-dimensional mesh. A more recent example of a system using a mesh is the ASCI Red supercomputer from the U.S. Department of Energy's Accelerated Strategic Computing Initiative, developed in 1995–97. ASCI Red, sited at Sandia National Laboratories, consists of 9,472 Pentium-II Xeon processors and uses a $38 \times 32 \times 2$ mesh interconnect for message passing. Meshes can also be used in shared memory systems.

Hypercube Network. In a d-dimensional (binary) *hypercube network*, each node connects to one node in each of the dimensions of the network. For example, in a three-dimensional hypercube, the connections in the x-direction, y-direction, and z-direction form a cube, as shown in Figure 1.10. Each node in a hypercube is assigned a d-bit binary address when there are d dimensions. Each bit is associated with one of the dimensions and can be a 0 or a 1, for the two nodes in that dimension. Nodes in a three-dimensional hypercube have a 3-bit address. Node 000 connects to nodes with addresses 001, 010, and 100. Node 111 connects to nodes 110, 101, and 011. Note that each node connects to nodes whose addresses differ by one bit. This characteristic can be extended for higher-dimension hypercubes. For example, in a five-dimensional hypercube, node 11101 connects to nodes 11100, 11111, 11001, 10101, and 01101.

A notable advantage of the hypercube is that the *diameter* of the network is given by $\log_2 p$ for a p-node hypercube, which has a reasonable (low) growth with increasing p. The

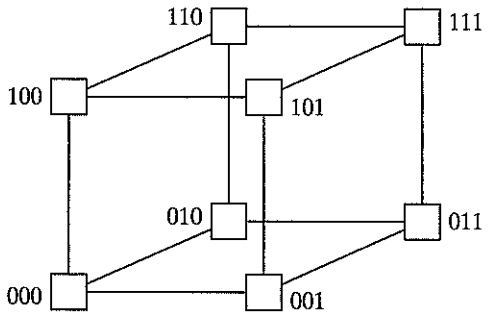

Figure 1.10 Three-dimensional hypercube.

number of links emanating from each node also only grows logarithmically. A very convenient aspect of the hypercube is the existence of a minimal distance deadlock-free routing algorithm. To describe this algorithm, let us route a message from a node X having a nodal address $X = x_{n-1}x_{n-2} \ldots x_1x_0$ to a destination node having a nodal address $Y = y_{n-1}y_{n-2} \cdots y_1y_0$. Each bit of Y that is different from that of X identifies one hypercube dimension that the route should take and can be found by performing the exclusive-OR function, $Z = X \oplus Y$, operating on pairs of bits. The dimensions to use in the routing are given by those bits of Z that are 1. At each node in the path, the exclusive-OR function between the current nodal address and the destination nodal address is performed. Usually the dimension identified by the most significant 1 in Z is chosen for the route. For example, the route taken from node 13 (001101) to node 42 (101010) in a six-dimensional hypercube would be node 13 (001101) to node 45 (101101) to node 41 (101001) to node 43 (101011) to node 42 (101010). This hypercube routing algorithm is sometimes called the *e-cube routing algorithm*, or *left-to-right routing*.

A d-dimensional hypercube actually consists of two $d-1$ dimensional hypercubes with dth dimension links between them. Figure 1.11 shows a four-dimensional hypercube drawn as two three-dimensional hypercubes with eight connections between them. Hence, the bisection width is 8. (The bisection width is $p/2$ for a p-node hypercube.) A five-dimensional hypercube consists of two four-dimensional hypercubes with connections between them, and so forth for larger hypercubes. In a practical system, the network must be laid out in two or possibly three dimensions.

Hypercubes are a part of a larger family of k-ary d-cubes; however, it is only the binary hypercube (with $k = 2$) that is really important as a basis for multicomputer construction and for parallel algorithms. The hypercube network became popular for constructing message-passing multicomputers after the pioneering research system called the Cosmic Cube was constructed at Caltech in the early 1980s (Seitz, 1985). However, interest in hypercubes has waned since the late 1980s.

As an alternative to direct links between individual computers, switches can be used in various configurations to route the messages between the computers.

Crossbar switch. The crossbar switch provides exhaustive connections using one switch for each connection. It is employed in shared memory systems more so than

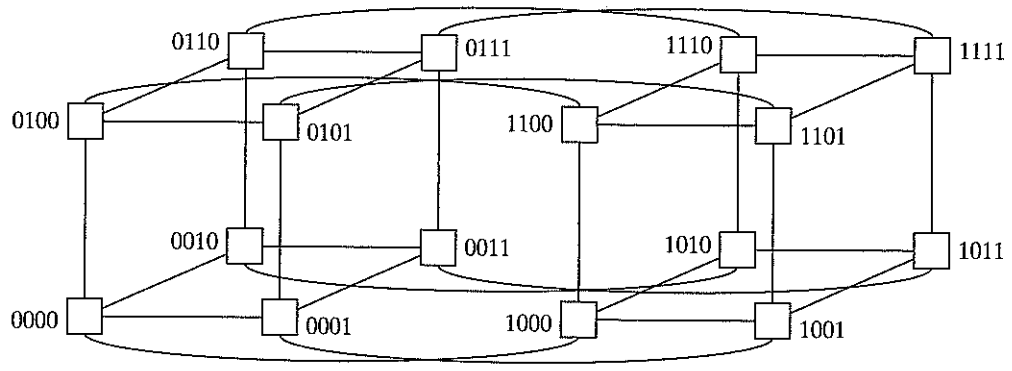

Figure 1.11 Four-dimensional hypercube.

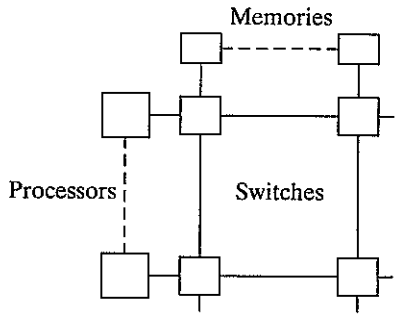

Figure 1.12 Cross-bar switch.

message-passing systems for connecting processor to memories. The layout of the crossbar switch is shown in Figure 1.12. There are several examples of systems using crossbar switches at some level with the system, especially very high performance systems. One of our students built a very early crossbar switch multiple microprocessor system in the 1970s (Wilkinson and Abachi, 1983).

Tree Networks. Another switch configuration is to use a *binary tree*, as shown in Figure 1.13. Each switch in the tree has two links connecting to two switches below it as the network fans out from the root. This particular tree is a *complete binary tree* because every level is fully occupied. The *height* of a tree is the number of links from the root to the lowest leaves. A key aspect of the tree structure is that the height is logarithmic; there are $\log_2 p$ levels of switches with p processors (at the leaves). The tree network need not be complete or based upon the base two. In an m-ary tree, each node connects to m nodes beneath it.

Under uniform request patterns, the communication traffic in a tree interconnection network increases toward the root, which can be a bottleneck. In a *fat tree network* (Leiserson, 1985), the number of the links is progressively increased toward the root. In a *binary fat tree*, we simply add links in parallel, as required between levels of a binary tree, and increase the number of links toward the root. Leiserson developed this idea into the *universal fat tree*, in which the number of links between nodes grows exponentially toward the root, thereby allowing increased traffic toward the root and reducing the communication bottleneck. The most notable example of a computer designed with tree interconnection networks is the Thinking Machine's Connection Machine CM5 computer, which uses a 4-ary fat tree (Hwang, 1993). The fat tree has been used subsequently. For example, the Quadrics QsNet network (see http://www.quadrics.com) uses a fat tree.

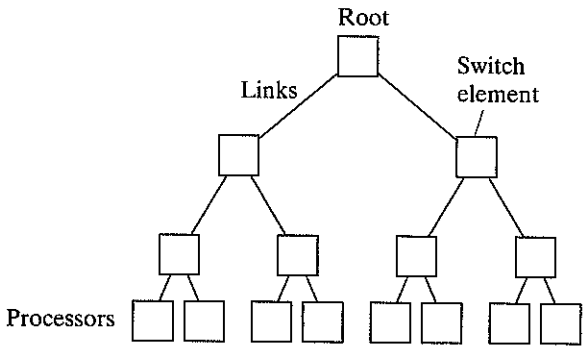

Figure 1.13 Tree structure.

Multistage Interconnection Networks. The multistage interconnection network (MIN) is a classification covering a multitude of configurations with the common characteristic of having a number of levels of switches. Switches in one level are connected to switches in adjacent levels in various symmetrical ways such that a path can made from one side of the network to the other side (and back sometimes). An example of a multistage interconnection network is the Omega network shown in Figure 1.14 (for eight inputs and outputs). This network has a very simple routing algorithm using the destination address. Inputs and outputs are given addresses as shown in the figure. Each switching cell requires one control signal to select either the upper output or the lower output (0 specifying the upper output and 1 specifying the lower). The most significant bit of the destination address is used to control the switch in the first stage; if the most significant bit is 0, the upper output is selected, and if it is 1, the lower output is selected. The next-most significant bit of the destination address is used to select the output of the switch in the next stage, and so on until the final output has been selected. The Omega network is highly blocking, though one path can always be made from any input to any output in a free network.

Multistage interconnection networks have a very long history and were originally developed for telephone exchanges, and are still sometimes used for interconnecting computers or groups of computers for really large systems. For example, the ASCI White supercomputer uses an Omega multistage interconnection network. For more information on multistage interconnection networks see Duato, Yalamanchili, and Ni (1997).

Communication Methods. The ideal situation in passing a message from a source node to a destination node occurs when there is a direct link between the source node and the destination node. In most systems and computations, it is often necessary to route a message through intermediate nodes from the source node to the destination node. There are two basic ways that messages can be transferred from a source to a destination: *circuit switching* and *packet switching*.

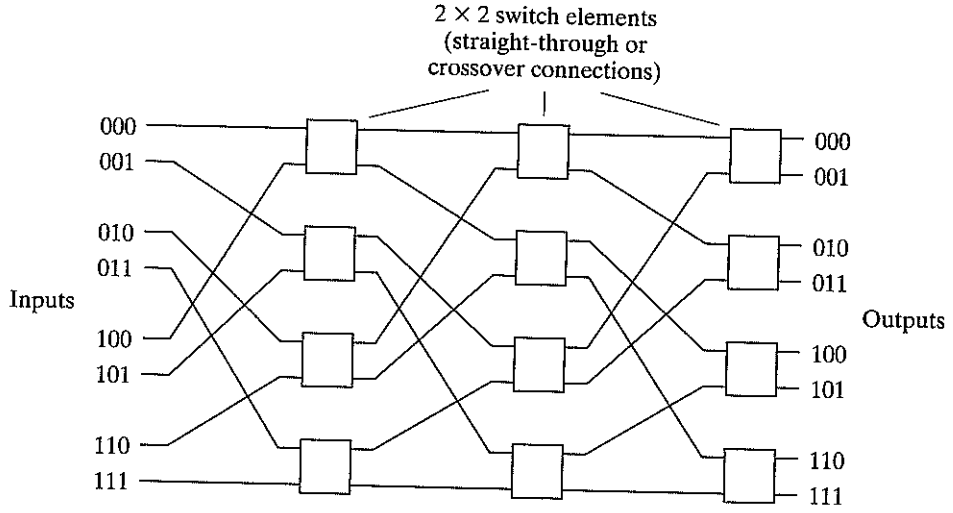

Figure 1.14 Omega network.

Circuit switching involves establishing the path and maintaining all the links in the path for the message to pass, uninterrupted, from the source to the destination. All the links are reserved for the transfer until the message transfer is complete. A simple telephone system (not using advanced digital techniques) is an example of a circuit-switched system. Once a telephone connection is made, the connection is maintained until the completion of the telephone call. Circuit switching has been used on some early multicomputers (e.g., the Intel IPSC-2 hypercube system), but it suffers from forcing all the links in the path to be reserved for the complete transfer. None of the links can be used for other messages until the transfer is completed.

In packet switching, the message is divided into "packets" of information, each of which includes the source and destination addresses for routing the packet through the interconnection network, and the data. There is a maximum size for the packet, say 1000 data bytes. If the message is larger than the maximum size, the message is broken up into separate packets, and each packet is sent through the network separately. Buffers are provided inside nodes to hold packets before they are transferred onward to the next node. A packet remains in a buffer if blocked from moving forward to the next node. The mail system is an example of a packet-switched system. Letters are moved from the mailbox to the post office and handled at intermediate sites before being delivered to the destination. This form of packet switching is called *store-and-forward packet switching* . Store-and-forward packet switching enables links to be used by other packets once the current packet has been forwarded. Unfortunately, store-and-forward packet switching, as described, incurs a significant latency, since packets must first be stored in buffers within each node, whether or not an outgoing link is available. This requirement is eliminated in *cut-through*, a technique originally developed for computer networks (Kermani and Kleinrock, 1979). In cut-through, if the outgoing link is available, the message is immediately passed forward without being stored in the nodal buffer; that is, it is "cut through." Thus, if the complete path were available, the message would pass immediately through to the destination. Note, however, that if the path is blocked, storage is needed for the complete message/packet being received.

Seitz introduced *wormhole* routing (Dally and Seitz, 1987) as an alternative to normal store-and-forward routing to reduce the size of the buffers and decrease the latency. In store-and-forward packet routing, a message is stored in a node and transmitted as a whole when an outgoing link becomes free. In wormhole routing, the message is divided into smaller units called *flits* (flow control digits). A flit is usually one or two bytes (Leighton, 1992). The link between nodes may provide for one wire for each bit in the flit so that the flit can be transmitted in parallel. Only the head of the message is initially transmitted from the source node to the next node when the connecting link is available. Subsequent flits of the message are transmitted when links become available, and the flits can be distributed through the network. When the head flit moves forward, the next one can move forward, and so on. A request/acknowledge signaling system is necessary between nodes to "pull" the flits along. When a flit is ready to move on from its buffer, it makes a request to the next node. When this node has a flit buffer empty, it calls for the flit from the sending node. It is necessary to reserve the complete path for the message as the parts of the message (the flits) are linked. Other packets cannot be interleaved with the flits along the same links.

Wormhole routing requires less storage at each node and produces a latency that is independent of the path length. Ni and McKinley (1993) present an analysis to show the independence of path length on latency in wormhole routing. If the length of a flit is much less than the total message, the latency of wormhole routing will be appropriately

constant irrespective of the length of the route. (Circuit switching will produce a similar characteristic.) In contrast, store-and-forward packet switching produces a latency that is approximately proportional to the length of the route, as is illustrated in Figure 1.15.

Interconnection networks, as we have seen, have routing algorithms to find a path between nodes. Some routing algorithms are adaptive in that they choose alternative paths through the network depending upon certain criteria, notably local traffic conditions. In general, routing algorithms, unless properly designed, can be prone to *livelock* and *deadlock. Livelock* can occur particularly in adaptive routing algorithms and describes the situation in which a packet keeps going around the network without ever finding its destination. *Deadlock* occurs when packets cannot be forwarded to the next node because they are blocked by other packets waiting to be forwarded, and these packets are blocked in a similar way so that none of the packets can move.

Deadlock can occur in both store-and-forward and wormhole networks. The problem of deadlock appears in communication networks using store-and-forward routing and has been studied extensively in that context. The mathematical conditions and solutions for deadlock-free routing in any network can be found in Dally and Seitz (1987). A general solution to deadlock is to provide *virtual* channels, each with separate buffers, for classes of messages. The *physical* links or channels are the actual hardware links between nodes. Multiple *virtual* channels are associated with a physical channel and time-multiplexed onto the physical channel, as shown in Figure 1.16. Dally and Seitz developed the use of separate virtual channels to avoid deadlock for wormhole networks.

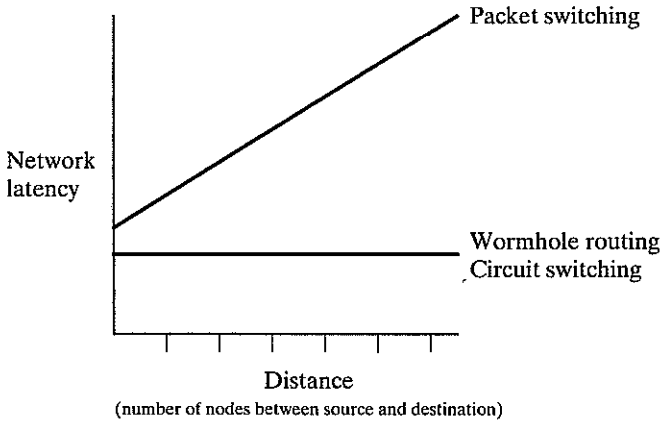

Figure 1.15 Network delay characteristics.

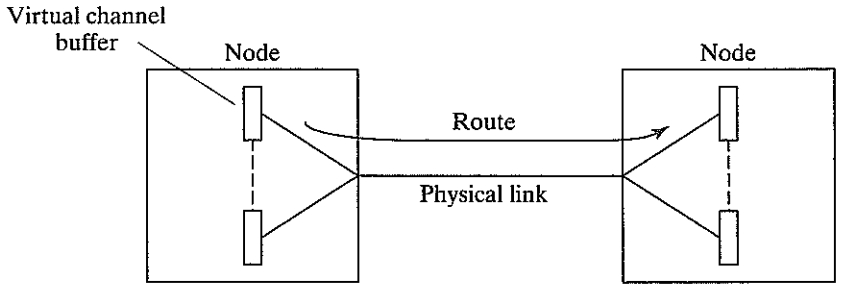

Figure 1.16 Multiple virtual channels mapped onto a single physical channel.

1.3.3 Distributed Shared Memory

The message-passing paradigm is often not as attractive for programmers as the shared memory paradigm. It usually requires the programmers to use explicit message-passing calls in their code, which is very error prone and makes programs difficult to debug. Message-passing programming has been compared to low-level assembly language programming (programming using the internal language of a processor). Data cannot be shared; it must be copied. This may be problematic in applications that require multiple operations across large amounts of data. However, the message-passing paradigm has the advantage that special synchronization mechanisms are not necessary for controlling simultaneous access to data. These synchronization mechanisms can significantly increase the execution time of a parallel program.

Recognizing that the shared memory paradigm is desirable from a programming point of view, several researchers have pursued the concept of a *distributed shared memory system.* As the name suggests, in a distributed shared memory system the memory is physically distributed with each processor, but each processor has access to the whole memory using a single memory address space. For a processor to access a location not in its local memory, message passing occurs to pass data between the processor and the memory location but in some automated way that hides the fact that the memory is distributed. Of course, accesses to remote locations will incur a greater delay, and usually a significantly greater delay, than for local accesses.

Multiprocessor systems can be designed in which the memory is physically distributed but operates as shared memory and appears from the programmer's perspective as shared memory. A number of projects have been undertaken to achieve this goal using specially designed hardware, and there have been commercial systems based upon this idea. Perhaps the most appealing approach is to use networked computers. One way to achieve distributed shared memory on a group of networked computers is to use the existing virtual memory management system of the individual computers which is already provided on almost all systems to manage its local memory hierarchy. The virtual memory management system can be extended to gives the illusion of global shared memory even when it is distributed in different computers.This idea is called *shared virtual memory.* One of the first to develop shared virtual memory was Li (1986). There are other ways to achieve distributed shared memory that do not require the use of the virtual memory management system or special hardware. In any event, physically the system is as given for message-passing multicomputers in Figure 1.8, except that now the local memory becomes part of the shared memory and is accessible from all processors, as illustrated in Figure 1.17.

Implementing and programming a distributed shared memory system is considered in detail in Chapter 9 after the fundamental concepts of shared memory programming in Chapter 8. Shared memory and message passing should be viewed as programming paradigms in that either could be the programming model for any type of multiprocessor, although specific systems may be designed for one or the other.

It should be mentioned that DSM implemented on top of a message-passing system usually will not have the performance of a true shared memory system, nor will using message-passing directly on a message system.

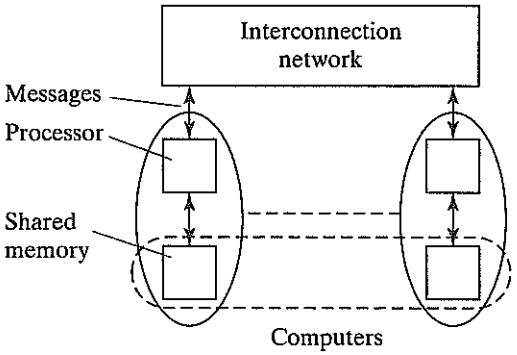

Figure 1.17 Shared memory multiprocessor.

1.3.4 MIMD and SIMD Classifications

In a single-processor computer, a single stream of instructions is generated by the program execution. The instructions operate upon data items. Flynn (1966) created a classification for computers and called this single-processor computer a *single instruction stream-single data stream* (SISD) computer. In a general-purpose multiprocessor system, each processor has a separate program, and one instruction stream is generated from each program for each processor. Each instruction operates upon different data. Flynn classified this type of computer as a *multiple instruction stream-multiple data stream* (MIMD) computer. The shared memory and message-passing multiprocessors so far described are both in the MIMD classification. The term MIMD has stood the test of time and is still widely used for a computer system operating in this mode.

Apart from the two extremes, SISD and MIMD, for certain applications there can be significant performance advantages in designing a computer in which a single instruction stream is from a single program but multiple data streams exist. The instructions from the program are broadcast to more than one processor. Each processor is essentially an arithmetic processor without a (program) control unit. A single control unit is responsible for fetching the instructions from memory and issuing the instructions to the processors. Each processor executes the same instruction in synchronism, but using different data. For flexibility, individual processors can be inhibited from participating in the instruction. The data items form an array, and an instruction acts upon the complete array in one instruction cycle. Flynn classified this type of computer as a *single instruction stream-multiple data stream* (SIMD) computer. The SIMD type of computer was developed because there are a number of important applications that mostly operate upon arrays of data. For example, most computer simulations of physical systems (from molecular systems to weather forecasting) start with large arrays of data points that must be manipulated. Another important application area is low-level image processing, in which the picture elements (pixels) of the image are stored and manipulated, as described in Chapter 12. Having a system that will perform similar operations on data points at the same time will be both efficient in hardware and relatively simple to program. The program simply consists of a single sequence of instructions operating on the array of data points together with normal control instructions executed by the separate control unit. We will not consider SIMD computers in this text as they are specially designed computers, often for specific applications. Computers today can have SIMD instructions for multimedia and graphics applications. For example, the

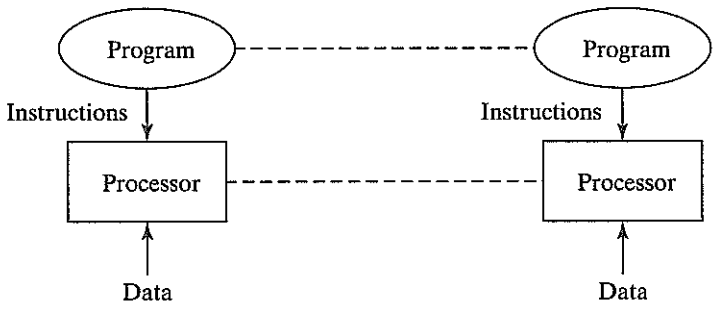

Figure 1.18 MPMD structure.

Pentium family, starting with the Pentium II, now has such SIMD instructions added to speed up multimedia and other applications that require the same operation to be performed on different data, the so-called MMX (MultiMedia eXtension) instructions.

The fourth combination of Flynn's classification, *multiple instruction stream-single data stream* (MISD) computer, does not exist unless one specifically classifies pipelined architectures in this group, or possibly some fault tolerant systems.

Within the MIMD classification, which we are concerned with, each processor has its own program to execute. This could be described as *multiple program multiple data* (MPMD) structure, as illustrated in Figure 1.18. Of course, all of the programs to be executed could be different, but typically only two source programs are written, one for a designated master processor and one for the remaining processors, which are called slave processors. A programming structure we may use, or may have to use, is the *single program multiple data* (SPMD) structure. In this structure, a single source program is written and each processor will execute its personal copy of this program, although independently and not in synchronism. The source program can be constructed so that parts of the program are executed by certain computers and not others depending upon the identity of the computer. For a master-slave structure, the program would have parts for the master and parts for the slaves.

1.4 CLUSTER COMPUTING

1.4.1 Interconnected Computers as a Computing Platform

So far, we have described specially designed parallel computers containing multiple processors or multiple computers as the computing platform for parallel computing. There have been numerous university research projects over the years designing such multiprocessor systems, often with radically different architectural arrangements and different software solutions, each project searching for the best performance. For large systems, the direct links have been replaced with switches and multiple levels of switches (multistage interconnection networks). Computer system manufacturers have come up with numerous designs. The major problem that most manufacturers have faced is the unending progress towards faster and faster processors. Each new generation of processors is faster and able to perform more simultaneous operations internally to boost performance. The most

obvious improvement noticed by the computer purchaser is the increase in the clock rate of personal computers. The basic clock rate continues to increase unabated. Imagine purchasing a Pentium (or any other) computer one year and a year later being able to purchase the same system but with twice the clock frequency. And in addition to clock rate, other factors make the system operate even faster. For example, newer designs may employ more internal parallelism within the processor and other ways to achieve faster operation. They often use memory configurations with higher bandwidth. The way around the problem of unending progress of faster processors for "supercomputer" manufacturers has been to use a huge number of available processors. For example, suppose a multiprocessor is designed with state-of-the-art processors in 2004, say 3GHz processors. Using 500 of these processors together should still overtake the performance of any single processor system for some years, but at an enormous cost.

In the late 1980s and early 1990s, another more cost-effective approach was tried by some universities—using workstations and personal computers connected together to form a powerful computing platform. A number of projects explored forming groups of computers from various perspectives. Some early projects explored using workstations as found in laboratories to form a *cluster of workstations* (COWs) or *network of workstations* (NOWs), such as the NOW project at Berkeley (Anderson, Culler, and Patterson, 1995). Some explored using the free time of existing workstations when they were not being used for other purposes, as oftentimes workstations, especially those in offices, are not used continuously or do not require 100% of the processor time even when they are being used.

Initially, using a network of workstations for parallel computing became interesting to many people because networks of workstations already existed for general-purpose computing. Workstations, as the name suggests, were already used for various programming and computer-related activities. It was quickly recognized that a network of workstations, offered a very attractive alternative to expensive supercomputers and parallel computer systems for high-performance computing. Using a network of workstations has a number of significant and well-enumerated advantages over specially designed multiprocessor systems. Key advantages are:

1. Very high performance workstations and PCs are readily available at low cost.
2. The latest processors can easily be incorporated into the system as they become available and the system can be expanded incrementally by adding additional computers, disks, and other resources.
3. Existing application software can be used or modified.

Software was needed to be able to use the workstations collectively, and fortuitously, at around the same time, message-passing tools were developed to make the concept usable. The most important message-passing project to provide parallel programming software tools for these workstations was Parallel Virtual Machine (PVM), started in the late 1980s. PVM was a key enabling technology and led to the success of using networks of workstations for parallel programming. Subsequently, the standard message-passing library, Message-Passing Interface (MPI), was defined.

The concept of using multiple interconnected personal computers (PCs) as a parallel computing platform matured in the 1990s as PCs became very inexpensive and powerful. Workstations, that is, computers particularly targeted towards laboratories, were being