

MODULE – 1

INTRODUCTION TO PARALLEL PROGRAMMING: PARALLEL HARDWARE AND PARALLEL SOFTWARE

It's perfectly feasible for specialists in disciplines other than computer science and computer engineering to write parallel programs. However, in order to write *efficient* parallel programs, we do need some knowledge of the underlying hardware and system software. It's also very useful to have some knowledge of different types of parallel software, so in this chapter we'll take a brief look at a few topics in hardware and software. We'll also take a brief look at evaluating program performance and a method for developing parallel programs. We'll close with a discussion of what kind of environment we might expect to be working in, and a few rules and assumptions we'll make in the rest of the book.

This is a long, broad chapter, so it may be a good idea to skim through some of the sections on a first reading so that you have a good idea of what's in the chapter. Then, when a concept or term in a later chapter isn't quite clear, it may be helpful to refer back to this chapter. In particular, you may want to skim over most of the material in "Modifications to the von Neumann Model," except "The Basics of Caching." Also, in the "Parallel Hardware" section, you can safely skim the material on "SIMD Systems" and "Interconnection Networks."

PARALLEL HARDWARE

Multiple issue and pipelining can clearly be considered to be parallel hardware, since functional units are replicated. However, since this form of parallelism isn't usually visible to the programmer, we're treating both of them as extensions to the basic von Neumann model, and for our purposes, parallel hardware will be limited to hardware that's visible to the programmer. In other words, if she can readily modify her source code to exploit it, or if she must modify her source code to exploit it, then we'll consider the hardware to be parallel.

1. SIMD systems

In parallel computing, **Flynn's taxonomy** is frequently used to classify computer architectures. It classifies a system according to the number of instruction streams and the

number of data streams it can simultaneously manage. A classical von Neumann system is therefore a **single instruction stream, single data stream**, or SISD system, since it executes a single instruction at a time and it can fetch or store one item of data at a time.

Single instruction, multiple data, or SIMD, systems are parallel systems. As the name suggests, SIMD systems operate on multiple data streams by applying the same instruction to multiple data items, so an abstract SIMD system can be thought of as having a single control unit and multiple ALUs. An instruction is broadcast from the control unit to the ALUs, and each ALU either applies the instruction to the current data item, or it is idle. As an example, suppose we want to carry out a “vector addition.” That is, suppose we have two arrays x and y , each with n elements, and we want to add the elements of y to the elements of x :

```
for (i = 0; i < n; i++)  
  x[i] += y[i];
```

Suppose further that our SIMD system has n ALUs. Then we could load $x[i]$ and $y[i]$ into the i th ALU, have the i th ALU add $y[i]$ to $x[i]$, and store the result in $x[i]$. If the system has m ALUs and $m < n$, we can simply execute the additions in blocks of m elements at a time. For example, if $m = 4$ and $n = 15$, we can first add elements 0 to 3, then elements 4 to 7, then elements 8 to 11, and finally elements 12 to 14. Note that in the last group of elements in our example—elements 12 to 14—we’re only operating on three elements of x and y , so one of the four ALUs will be idle.

The requirement that all the ALUs execute the same instruction or are idle can seriously degrade the overall performance of a SIMD system. For example, suppose we only want to carry out the addition if $y[i]$ is positive:

```
for (i = 0; i < n; i++)  
  
  if (y[i] > 0.0) x[i] += y[i];
```

In this setting, we must load each element of y into an ALU and determine whether it’s positive. If $y[i]$ is positive, we can proceed to carry out the addition. Otherwise, the ALU storing $y[i]$ will be idle while the other ALUs carry out the addition.

Note also that in a “classical” SIMD system, the ALUs must operate syn-chronously, that is, each ALU must wait for the next instruction to be broadcast before proceeding. Further, the ALUs have no instruction storage, so an ALU can’t delay execution of an instruction by storing it for later execution.

Finally, as our first example shows, SIMD systems are ideal for parallelizing sim-ple loops that operate on large arrays of data. Parallelism that’s obtained by dividing data among the processors and having the processors all apply (more or less) the same instructions to their subsets of the data is called **data-parallelism**. SIMD parallelism can be very efficient on large data parallel problems, but SIMD systems often don’t do very well on other types of parallel problems.

SIMD systems have had a somewhat checkered history. In the early 1990s a maker of SIMD systems (Thinking Machines) was the largest manufacturer of par-allel supercomputers. However, by the late 1990s the only widely produced SIMD systems were **vector processors**. More recently, graphics processing units, or GPUs, and desktop CPUs are making use of aspects of SIMD computing.

Vector processors

Although what constitutes a vector processor has changed over the years, their key characteristic is that they can operate on arrays or *vectors* of data, while conventional CPUs operate on individual data elements or *scalars*. Typical recent systems have the following characteristics:

Vector registers. These are registers capable of storing a vector of operands and operating simultaneously on their contents. The vector length is fixed by the system, and can range from 4 to 128 64-bit elements.

Vectorized and pipelined functional units. Note that the same operation is applied

to each element in the vector, or, in the case of operations like addition, the same operation is applied to each pair of corresponding elements in the two vectors.

Thus, vector operations are SIMD.

Vector instructions. These are instructions that operate on vectors rather than scalars. If the vector length is vector length, these instructions have the great virtue that a simple loop such as

```
for (i = 0; i < n; i++)  
x[i] += y[i];
```

requires only a single load, add, and store for each block of vector length elements, while a conventional system requires a load, add, and store for each

element. *Interleaved memory.* The memory system consists of multiple “banks” of memory, which can be accessed more or less independently. After accessing one bank, there will be a delay before it can be reaccessed, but a different bank can be accessed

much sooner. So if the elements of a vector are distributed across multiple banks, there can be little to no delay in loading/storing successive elements.

Strided memory access and hardware scatter/gather. In *strided memory access*, the program accesses elements of a vector located at fixed intervals. For example, accessing the first element, the fifth element, the ninth element, and so on, would be strided access with a stride of four. Scatter/gather (in this context) is writing (scatter) or reading (gather) elements of a vector located at irregular intervals— for example, accessing the first element, the second element, the fourth element, the eighth element, and so on. Typical vector systems provide special hardware to accelerate strided access and scatter/gather.

Vector processors have the virtue that for many applications, they are very fast and very easy to use. Vectorizing compilers are quite good at identifying code that can be vectorized. Further, they identify loops that cannot be vectorized, and they often provide information about why a loop couldn't be vectorized. The user can thereby make informed decisions about whether it's possible to rewrite the loop so that it will vectorize. Vector systems have very high memory bandwidth, and every data item that's loaded is actually used, unlike cache-based systems that may not make use of every item in a cache line. On the other hand, they don't handle irregular data structures as well as other parallel architectures, and there seems to be a very finite limit to their **scalability**, that is, their ability to handle ever larger problems. It's difficult to see how systems could be created that would operate on ever longer vectors. Current generation

systems scale by increasing the number of vector processors, not the vector length. Current commodity systems provide limited support for operations on very short vectors, while processors that operate on long vectors are custom manufactured, and, consequently, very expensive.

Graphics processing units

Real-time graphics application programming interfaces, or APIs, use points, lines, and triangles to internally represent the surface of an object. They use a **graphics pro-cessing pipeline** to convert the internal representation into an array of pixels that can

be sent to a computer screen. Several of the stages of this pipeline are programmable. The behavior of the programmable stages is specified by functions called **shader functions**. The shader functions are typically quite short—often just a few lines of C code. They're also implicitly parallel, since they can be applied to multiple elements (e.g., vertices) in the graphics stream. Since the application of a shader function to nearby elements often results in the same flow of control, GPUs can optimize performance by using SIMD parallelism, and in the current generation all GPUs use SIMD parallelism. This is obtained by including a large number of ALUs (e.g., 80) on each GPU processing core.

Processing a single image can require very large amounts of data—hundreds of megabytes of data for a single image is not unusual. GPUs therefore need to maintain very high rates of data movement, and in order to avoid stalls on memory accesses, they rely heavily on hardware multithreading; some systems are capable of storing the state of more than a hundred suspended threads for each executing thread. The actual number of threads depends on the amount of resources (e.g., registers) needed by the shader function. A drawback here is that many threads processing a lot of data are needed to keep the ALUs busy, and GPUs may have relatively poor performance on small problems.

It should be stressed that GPUs are not pure SIMD systems. Although the ALUs on a given core do use SIMD parallelism, current generation GPUs can have dozens of cores, which are capable of executing independent instruction streams.

GPUs are becoming increasingly popular for general, high-performance computing, and

several languages have been developed that allow users to exploit their power. For further details see [30].

2. MIMD systems

Multiple instruction, multiple data, or MIMD, systems support multiple simultaneous instruction streams operating on multiple data streams. Thus, MIMD systems typically consist of a collection of fully independent processing units or cores, each of which has its own control unit and its own ALU. Furthermore, unlike SIMD systems, MIMD systems are usually **asynchronous**, that is, the processors can operate at their own pace. In many MIMD systems there is no global clock, and there may be no relation between the system times on two different processors. In fact, unless the programmer imposes some synchronization, even if the processors are executing exactly the same sequence of instructions, at any given instant they may be executing different statements.

As we noted in Chapter 1, there are two principal types of MIMD systems: shared-memory systems and distributed-memory systems. In a **shared-memory system** a collection of autonomous processors is connected to a memory system via an interconnection network, and each processor can access each memory location. In a shared-memory system, the processors usually communicate implicitly by accessing shared data structures. In a **distributed-memory system**, each processor is paired with its own *private* memory, and the processor-memory pairs communicate over an interconnection network. So in distributed-memory systems the processors usually communicate explicitly by sending messages or by using special functions that provide access to the memory of another processor. See Figures 2.3 and 2.4.

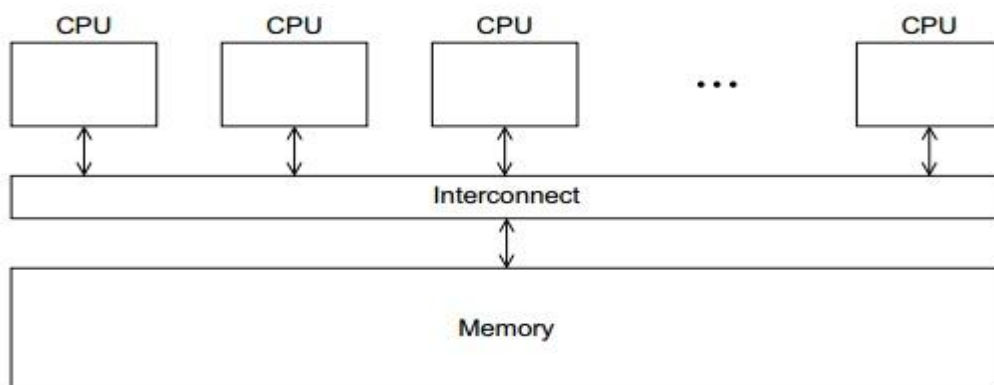
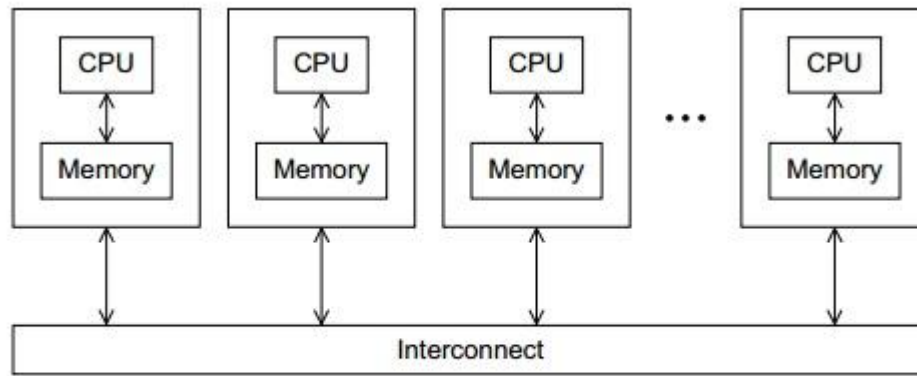


FIGURE 2.3

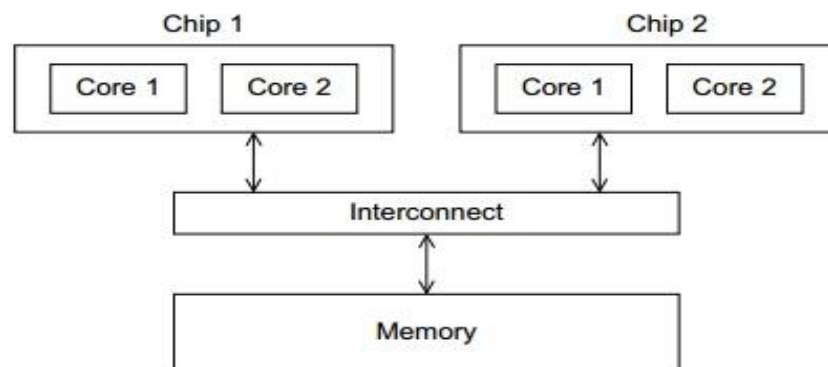
A shared-memory system

**FIGURE 2.4**

A distributed-memory system

Shared-memory systems

The most widely available shared-memory systems use one or more **multicore** processors. As we discussed in Chapter 1, a multicore processor has multiple CPUs or cores on a single chip. Typically, the cores have private level 1 caches, while other caches may or may not be shared between the cores.

**FIGURE 2.5**

A UMA multicore system

In shared-memory systems with multiple multicore processors, the interconnect can either connect all the processors directly to main memory or each processor can have a direct connection to a block of main memory, and the processors can access each others' blocks of main memory through special hardware built into the processors. See Figures 2.5 and 2.6. In the first type of system, the time to access all the memory locations will be the same for all the cores, while in the second type a memory location to which a core is directly connected can be

accessed more quickly than a memory location that must be accessed through another chip. Thus, the first type of system is called a **uniform memory access**, or UMA, system, while the second type is called a **nonuniform memory access**, or NUMA, system. UMA systems are usually easier to program, since the programmer doesn't need to worry about different access times for different memory locations. This advantage can be offset by the faster access to the directly connected memory in NUMA systems. Furthermore, NUMA systems have the potential to use larger amounts of memory than UMA systems.

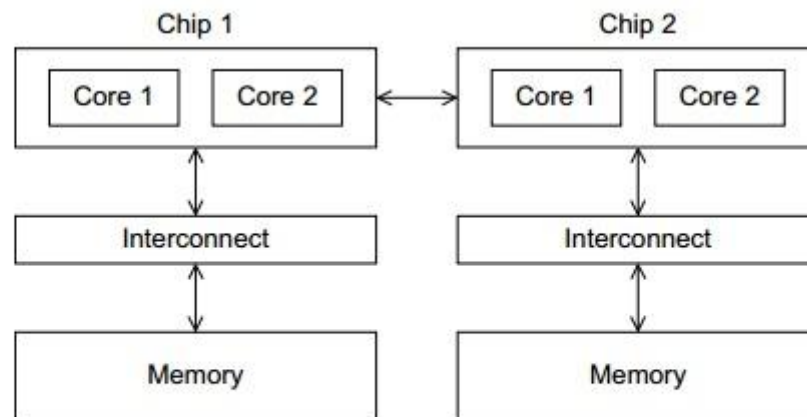


FIGURE 2.6

A NUMA multicore system

Distributed-memory systems

The most widely available distributed-memory systems are called **clusters**. They are composed of a collection of commodity systems—for example, PCs—connected by a commodity interconnection network—for example, Ethernet. In fact, the **nodes** of these systems, the individual computational units joined together by the communication network, are usually shared-memory systems with one or more multicore processors. To distinguish such systems from pure distributed-memory systems, they are sometimes called **hybrid systems**. Nowadays, it's usually understood that a cluster will have shared-memory nodes.

The **grid** provides the infrastructure necessary to turn large networks of geographically distributed computers into a unified distributed-memory system. In general, such a system will be *heterogeneous*, that is, the individual nodes may be built from different types of hardware.

3. Interconnection networks

The interconnect plays a decisive role in the performance of both distributed- and shared-memory systems: even if the processors and memory have virtually unlimited performance, a slow interconnect will seriously degrade the overall performance of all but the simplest parallel program. See, for example, Exercise 2.10.

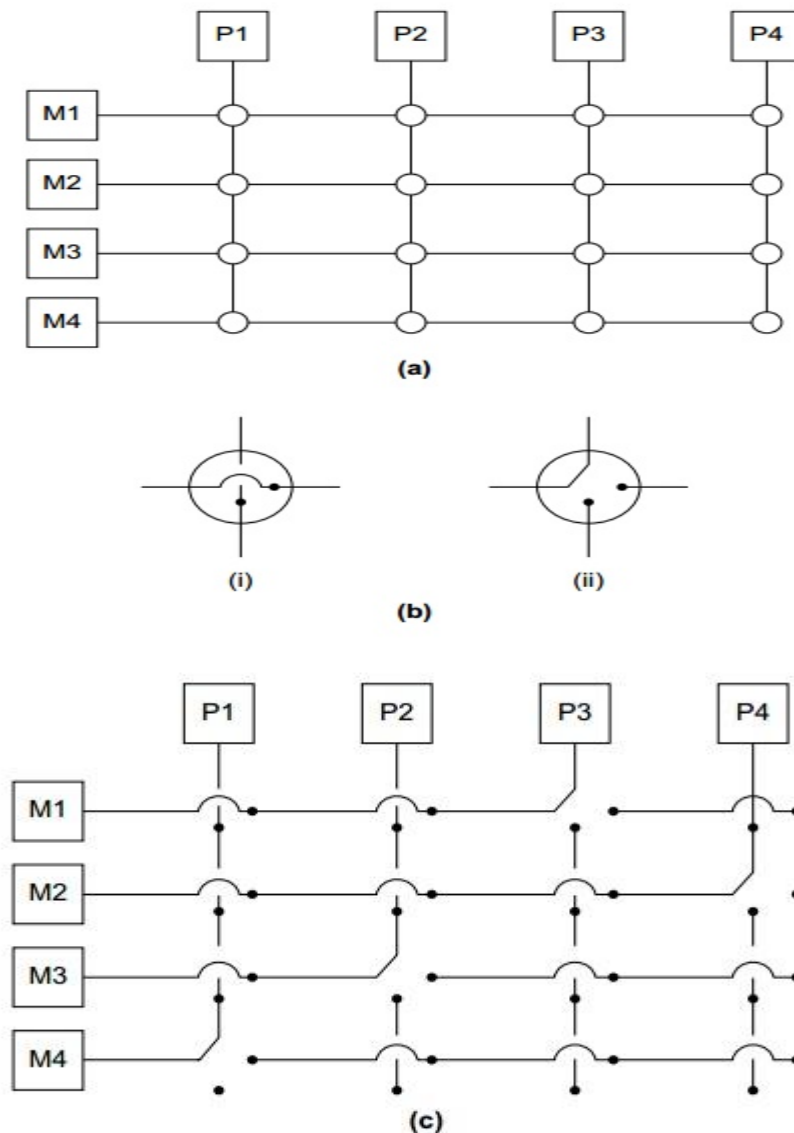
Although some of the interconnects have a great deal in common, there are enough differences to make it worthwhile to treat interconnects for shared-memory and distributed-memory separately.

Shared-memory interconnects

Currently the two most widely used interconnects on shared-memory systems are buses and crossbars. Recall that a **bus** is a collection of parallel communication wires together with some hardware that controls access to the bus. The key characteristic of a bus is that the communication wires are shared by the devices that are connected to it. Buses have the virtue of low cost and flexibility; multiple devices can be connected to a bus with little additional cost. However, since the communication wires are shared, as the number of devices connected to the bus increases, the likelihood that there will be contention for use of the bus increases, and the expected performance of the bus decreases. Therefore, if we connect a large number of processors to a bus, we would expect that the processors would frequently have to wait for access to main memory. Thus, as the size of shared-memory systems increases, buses are rapidly being replaced by *switched* interconnects.

As the name suggests, **switched** interconnects use switches to control the routing of data among the connected devices. A **crossbar** is illustrated in Figure 2.7(a). The lines are bidirectional communication links, the squares are cores or memory modules, and the circles are switches.

The individual switches can assume one of the two configurations shown in Figure 2.7(b). With these switches and at least as many memory modules as processors, there will only be a conflict between two cores attempting to access memory

**FIGURE 2.7**

(a) A crossbar switch connecting four processors (P_i) and four memory modules (M_j); (b) configuration of internal switches in a crossbar; (c) simultaneous memory accesses by the processors

if the two cores attempt to simultaneously access the same memory module. For example, Figure 2.7(c) shows the configuration of the switches if P_1 writes to M_4 , P_2 reads from M_3 , P_3 reads from M_1 , and P_4 writes to M_2 .

Crossbars allow simultaneous communication among different devices, so they are much faster than buses. However, the cost of the switches and links is relatively high. A small bus-based system will be much less expensive than a crossbar-based system of the same size.

Distributed-memory interconnects

Distributed-memory interconnects are often divided into two groups: direct inter-connects and indirect interconnects. In a **direct interconnect** each switch is directly connected to a processor-memory pair, and the switches are connected to each other. Figure 2.8 shows a **ring** and a two-dimensional **toroidal mesh**. As before, the circles are switches, the squares are processors, and the lines are bidirectional links. A ring is superior to a simple bus since it allows multiple simultaneous communications. However, it's easy to devise communication schemes in which some of the processors must wait for other processors to complete their communications. The toroidal mesh will be more expensive than the ring, because the switches are more complex—they must support five links instead of three—and if there are p processors, the number of links is $3p$ in a toroidal mesh, while it's only $2p$ in a ring. However, it's not difficult to convince yourself that the number of possible simultaneous communications patterns is greater with a mesh than with a ring.

One measure of “number of simultaneous communications” or “connectivity” is **bisection width**. To understand this measure, imagine that the parallel system is

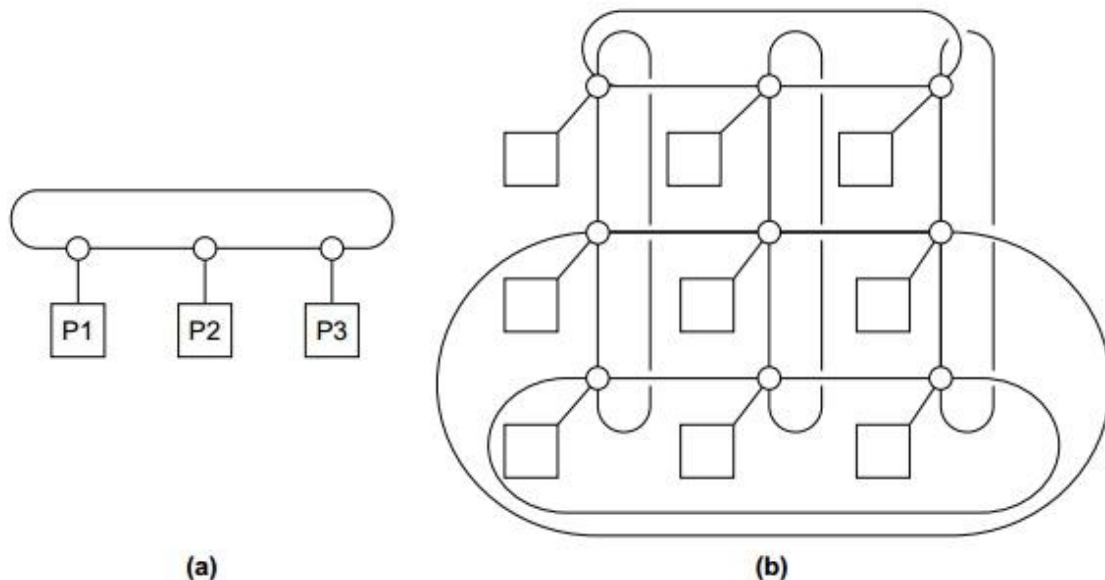
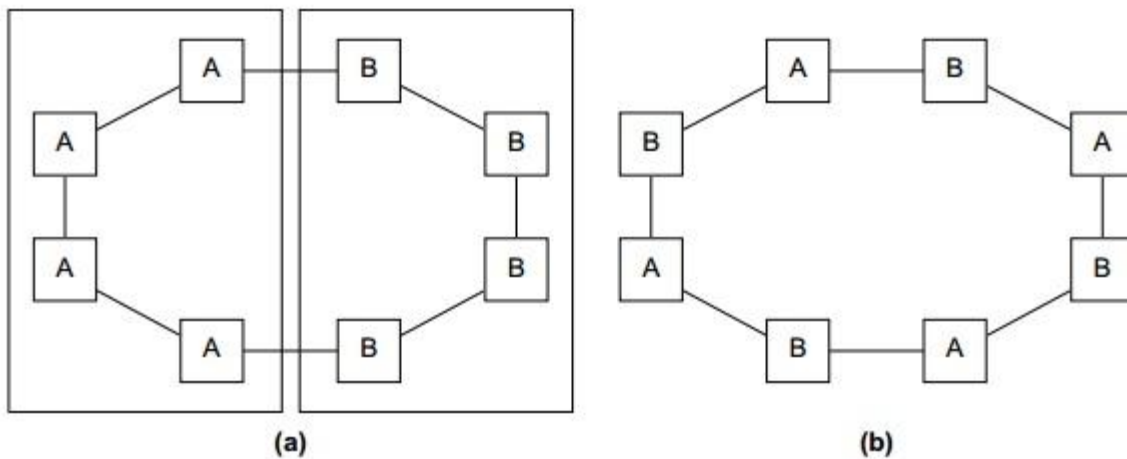


FIGURE 2.8

(a) A ring and (b) a toroidal mesh

**FIGURE 2.9**

Two bisections of a ring: (a) only two communications can take place between the halves and (b) four simultaneous connections can take place

divided into two halves, and each half contains half of the processors or nodes. How many simultaneous communications can take place “across the divide” between the halves? In Figure 2.9(a) we’ve divided a ring with eight nodes into two groups of four nodes, and we can see that only two communications can take place between the halves. (To make the diagrams easier to read, we’ve grouped each node with its switch in this and subsequent diagrams of direct interconnects.) However, in Figure 2.9(b) we’ve divided the nodes into two parts so that four simultaneous communications can take place, so what’s the bisection width? The bisection width is supposed to give a “worst-case” estimate, so the bisection width is two—not four.

An alternative way of computing the bisection width is to remove the minimum number of links needed to split the set of nodes into two equal halves. The number of links removed is the bisection width. If we have a square two-dimensional toroidal mesh with $p = q^2$ nodes (where q is even), then we can split the nodes into two halves by removing the “middle” horizontal links and the “wraparound” horizontal links. See Figure 2.10. This suggests that the bisection width is at most $2q = 2^{\text{Root}}p$. In fact, this is the smallest possible number of links and the bisection width of a square two-dimensional toroidal mesh is $2^{\text{Root}}p$.

The **bandwidth** of a link is the rate at which it can transmit data. It’s usually given in megabits or megabytes per second. **Bisection bandwidth** is often used as a measure of network quality. It’s similar to bisection width. However, instead of counting the number of links joining the halves, it sums the bandwidth of the links. For example, if the links in a ring have a bandwidth

of one billion bits per second, then the bisection bandwidth of the ring will be two billion bits per second or 2000 megabits per second.

The ideal direct interconnect is a **fully connected network** in which each switch is directly connected to every other switch. See Figure 2.11. Its bisection width is $p^2=4$. However, it's impractical to construct such an interconnect for systems with more than a few nodes, since it requires a total of $p^2=2 + p=2$ links, and each switch must be capable of connecting to p links. It is therefore more a “theoretical best possible” interconnect than a practical one, and it is used as a basis for evaluating other interconnects.

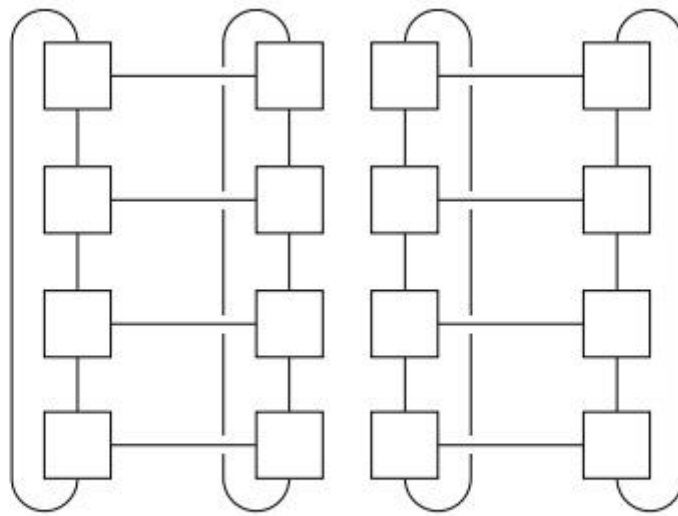


FIGURE 2.10

A bisection of a toroidal mesh

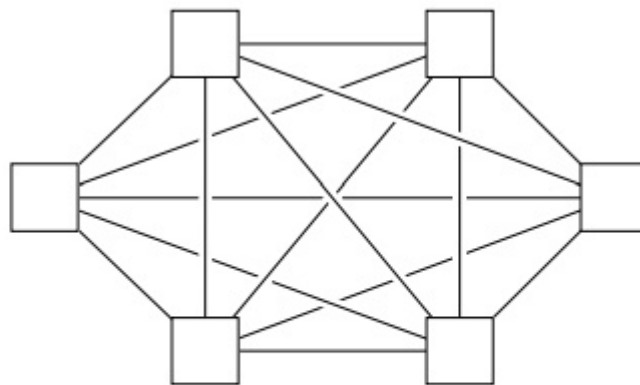
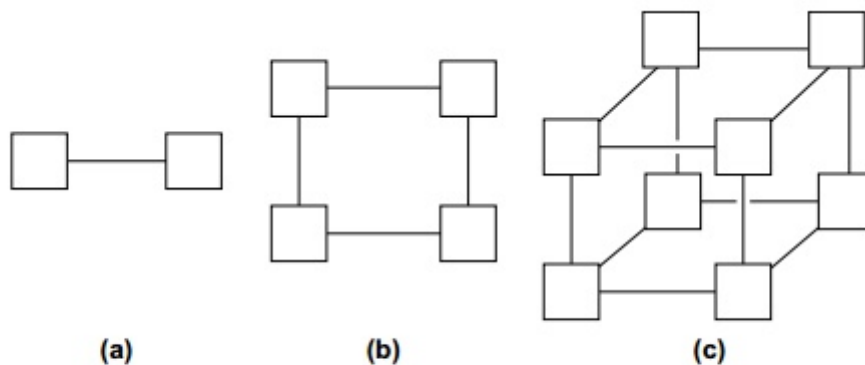


FIGURE 2.11

A fully connected network

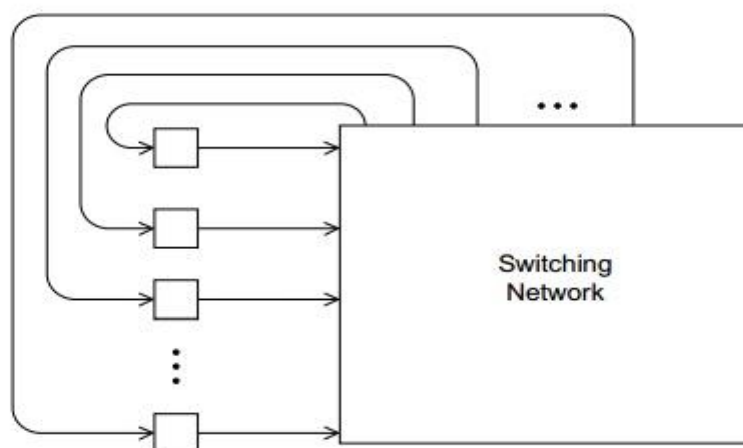
The **hypercube** is a highly connected direct interconnect that has been used in actual systems. Hypercubes are built inductively: A one-dimensional hypercube is a fully-connected system

with two processors. A two-dimensional hypercube is built from two one-dimensional hypercubes by joining “corresponding” switches. Similarly, a three-dimensional hypercube is built from two two-dimensional hypercubes. See Figure 2.12. Thus, a hypercube of dimension d has $p = 2^d$ nodes, and a switch in a d -dimensional hypercube is directly connected to a processor and d switches. The bisection width of a hypercube is $p/2$, so it has more connectivity than a ring or toroidal mesh, but the switches must be more powerful, since they must support $1 + d = 1 + \log_2 p$ wires, while the mesh switches only require five wires. So a hypercube with p nodes is more expensive to construct than a toroidal mesh.

**FIGURE 2.12**

(a) One-, (b) two-, and (c) three-dimensional hypercubes

Indirect interconnects provide an alternative to direct interconnects. In an indirect interconnect, the switches may not be directly connected to a processor. They're often shown with unidirectional links and a collection of processors, each of which has an outgoing and an incoming link, and a switching network. See Figure 2.13.

**FIGURE 2.13**

A generic indirect network

The **crossbar** and the **omega network** are relatively simple examples of indirect networks. We saw a shared-memory crossbar with bidirectional links earlier (Figure 2.7). The diagram of a distributed-memory crossbar in Figure 2.14 has unidirectional links. Notice that as long as two processors don't attempt to communicate with the same processor, all the processors can simultaneously communicate with another processor.

An omega network is shown in Figure 2.15. The switches are two-by-two cross-bars (see Figure 2.16). Observe that unlike the crossbar, there are communications that cannot occur simultaneously. For example, in Figure 2.15 if processor 0 sends

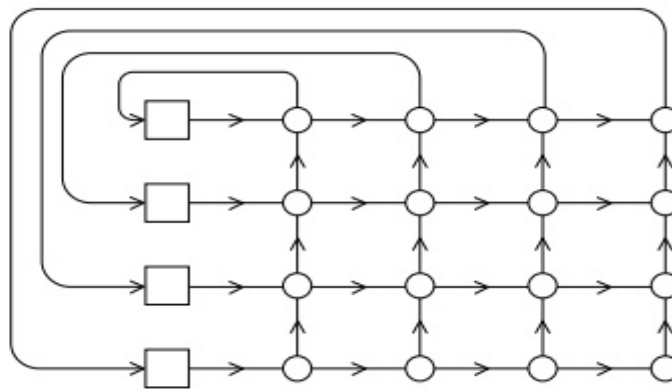


FIGURE 2.14

A crossbar interconnect for distributed-memory

a message to processor 6, then processor 1 cannot simultaneously send a message to processor 7. On the other hand, the omega network is less expensive than the crossbar. The omega network uses $\frac{1}{2} p \log_2(p)$ of the 2×2 crossbar switches, so it uses a total of $2p \log_2(p)$ switches, while the crossbar uses p^2 .

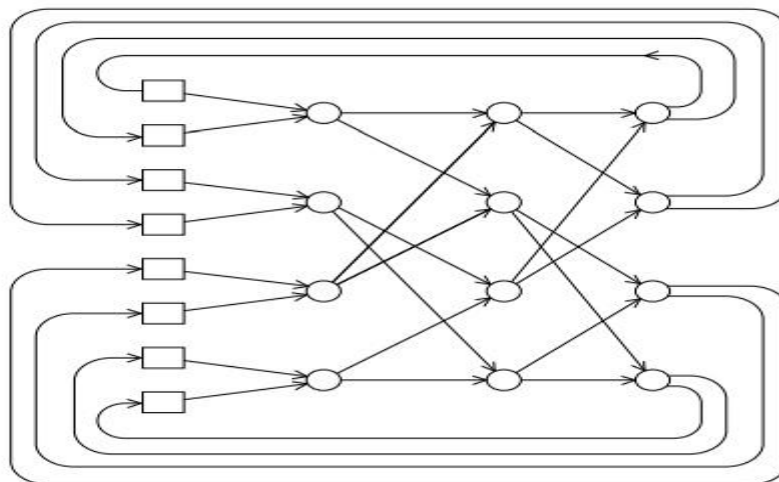
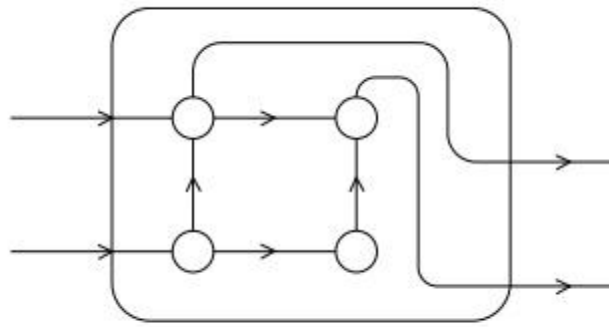


FIGURE 2.15

An omega network

**FIGURE 2.16**

A switch in an omega network

It's a little bit more complicated to define bisection width for indirect networks. See Exercise 2.14. However, the principle is the same: we want to divide the nodes into two groups of equal size and determine how much communication can take place between the two halves, or alternatively, the minimum number of links that need to be removed so that the two groups can't communicate. The bisection width of a $p \times p$ crossbar is p and the bisection width of an omega network is $p/2$.

Latency and bandwidth

Any time data is transmitted, we're interested in how long it will take for the data to reach its destination. This is true whether we're talking about transmitting data between main memory and cache, cache and register, hard disk and memory, or between two nodes in a distributed-memory or hybrid system. There are two figures that are often used to describe the performance of an interconnect (regardless of what it's connecting): the **latency** and the **bandwidth**. The latency is the time that elapses between the source's beginning to transmit the data and the destination's starting to receive the first byte. The bandwidth is the rate at which the destination receives data after it has started to receive the first byte. So if the latency of an interconnect is l seconds and the bandwidth is b bytes per second, then the time it takes to transmit a message of n bytes is

$$\text{message transmission time} = l + n/b.$$

Beware, however, that these terms are often used in different ways. For example, latency is sometimes used to describe total message transmission time. It's also often used to describe

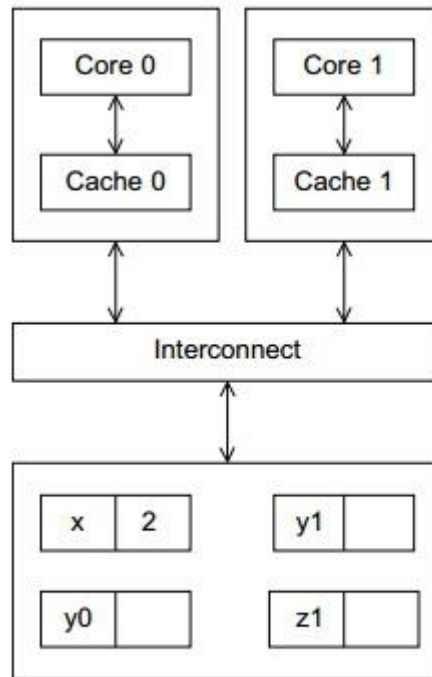
the time required for any fixed overhead involved in transmitting data. For example, if we're sending a message between two nodes in a distributed-memory system, a message is not just raw data. It might include the data to be transmitted, a destination address, some information specifying the size of the message, some information for error correction, and so on. So in this setting, latency might be the time it takes to assemble the message on the sending side—the time needed to combine the various parts—and the time to disassemble the message on the receiving side—the time needed to extract the raw data from the message and store it in its destination.

4. Cache coherence

Recall that CPU caches are managed by system hardware: programmers don't have direct control over them. This has several important consequences for shared-memory systems. To understand these issues, suppose we have a shared-memory system with two cores, each of which has its own private data cache. See Figure 2.17. As long as the two cores only read shared data, there is no problem. For example, suppose that x is a shared variable that has been initialized to 2, y_0 is private and owned by core 0, and y_1 and z_1 are private and owned by core 1. Now suppose the following statements are executed at the indicated times:

| Time | Core 0 | Core 1 |
|------|--------------------------------|--------------------------------|
| 0 | $y_0 = x;$ | $y_1 = 3 * x;$ |
| 1 | $x = 7;$ | Statement(s) not involving x |
| 2 | Statement(s) not involving x | $z_1 = 4 * x;$ |

Then the memory location for y_0 will eventually get the value 2, and the memory location for y_1 will eventually get the value 6. However, it's not so clear what value z_1 will get. It might at first appear that since core 0 updates x to 7 before the assignment to z_1 , z_1 will get the value $4 * 7 = 28$. However, at time 0, x is in the cache of core 1. So unless for some reason x is evicted from core 0's cache and then reloaded into core 1's cache, it actually appears that the original value $x = 2$ may be used, and z_1 will get the value $4 * 2 = 8$.

**FIGURE 2.17**

A shared-memory system with two cores and two caches

Note that this unpredictable behavior will occur regardless of whether the system is using a write-through or a write-back policy. If it's using a write-through policy, the main memory will be updated by the assignment $x = 7$. However, this will have no effect on the value in the cache of core 1. If the system is using a write-back policy, the new value of x in the cache of core 0 probably won't even be available to core 1 when it updates $z1$.

Clearly, this is a problem. The programmer doesn't have direct control over when the caches are updated, so her program cannot execute these apparently innocuous statements and know what will be stored in $z1$. There are several problems here, but the one we want to look at right now is that the caches we described for single processor systems provide no mechanism for insuring that when the caches of multiple processors store the same variable, an update by one processor to the cached variable is "seen" by the other processors. That is, that the cached value stored by the other processors is also updated. This is called the **cache coherence** problem.

Snooping cache coherence

There are two main approaches to insuring cache coherence: **snooping cache coherence** and directory-based cache coherence. The idea behind snooping comes from bus-based systems: When the cores share a bus, any signal transmitted on the bus can be "seen" by all the cores

connected to the bus. Thus, when core 0 updates the copy of x stored in its cache, if it also broadcasts this information across the bus, and if core 1 is “snooping” the bus, it will see that x has been updated and it can mark its copy of x as invalid. This is more or less how snooping cache coherence works. The principal difference between our description and the actual snooping protocol is that the broadcast only informs the other cores that the *cache line* containing x has been updated, not that x has been updated.

A couple of points should be made regarding snooping. First, it’s not essential that the interconnect be a bus, only that it support broadcasts from each processor to all the other processors. Second, snooping works with both write-through and write-back caches. In principle, if the interconnect is shared—as with a bus—with write-through caches there’s no need for additional traffic on the interconnect, since each core can simply “watch” for writes. With write-back caches, on the other hand, an extra communication *is* necessary, since updates to the cache don’t get immediately sent to memory.

Directory-based cache coherence

Unfortunately, in large networks broadcasts are expensive, and snooping cache coherence requires a broadcast every time a variable is updated (but see Exercise 2.15). So snooping cache coherence isn’t scalable, because for larger systems it will cause performance to degrade. For example, suppose we have a system with the basic distributed-memory architecture (Figure 2.4). However, the system provides a single address space for all the memories. So, for example, core 0 can access the variable x stored in core 1’s memory, by simply executing a statement such as $y = x$.

(Of course, accessing the memory attached to another core will be slower than accessing “local” memory, but that’s another story.) Such a system can, in principle, scale to very large numbers of cores. However, snooping cache coherence is clearly a problem since a broadcast across the interconnect will be very slow relative to the speed of accessing local memory.

Directory-based cache coherence protocols attempt to solve this problem through the use of a data structure called a **directory**. The directory stores the status of each cache line. Typically, this data structure is distributed; in our example, each core/memory pair might be responsible for storing the part of the structure that specifies the status of the cache lines in its local memory. Thus, when a line is read into, say, core 0’s cache, the directory entry corresponding to that line would be updated indicating that core 0 has a copy of the line. When a variable is

updated, the directory is consulted, and the cache controllers of the cores that have that variable's cache line in their caches are invalidated.

Clearly there will be substantial additional storage required for the directory, but when a cache variable is updated, only the cores storing that variable need to be contacted.

False sharing

It's important to remember that CPU caches are implemented in hardware, so they operate on cache lines, not individual variables. This can have disastrous consequences for performance. As an example, suppose we want to repeatedly call a function $f(i,j)$ and add the computed values into a vector:

```
int i, j, m, n; double y[m];

/*          Assign y = 0          */
...
for (i = 0; i < m; i++) for (j = 0; j < n; j++)
y[i] += f(i,j);
```

We can parallelize this by dividing the iterations in the outer loop among the cores. If we have `core count` cores, we might assign the first `m/core count` iterations to the first core, the next `m/core count` iterations to the second core, and so on.

```
/* Private variables */
int i, j, iter count;

/ Shared variables initialized by one core */
int m, n, core count

double y[m];

iter count = m/core count

/* Core 0 does this */
```

```
for (i = 0; i < iter count; i++) for (j = 0; j < n; j++)  
y[i] += f(i,j);
```

```
/* Core 1 does this */
```

```
for (i = iter count+1; i < 2 iter count; i++) for (j = 0; j < n; j++)
```

```
y[i] += f(i,j);
```

```
...
```

Now suppose our shared-memory system has two cores, $m = 8$, doubles are eight bytes, cache lines are 64 bytes, and $y[0]$ is stored at the beginning of a cache line. A cache line can store eight doubles, and y takes one full cache line. What happens when core 0 and core 1 simultaneously execute their codes? Since all of y is stored in a single cache line, each time one of the cores executes the statement $y[i] += f(i,j)$, the line will be invalidated, and the next time the other core tries to execute this statement it will have to fetch the updated line from memory! So if n is large, we would expect that a large percentage of the assignments $y[i] += f(i,j)$ will access main memory—in spite of the fact that core 0 and core 1 never access each others' elements of y . This is called **false sharing**, because the system is behaving *as if* the elements of y were being shared by the cores.

Note that false sharing does not cause incorrect results. However, it can ruin the performance of a program by causing many more accesses to memory than necessary. We can reduce its effect by using temporary storage that is local to the thread or process and then copying the temporary storage to the shared storage. We'll return to the subject of false sharing in Chapters 4 and 5.

5. Shared-memory versus distributed-memory

Newcomers to parallel computing sometimes wonder why all MIMD systems aren't shared-memory, since most programmers find the concept of implicitly coordinating the work of the processors through shared data structures more appealing than explicitly sending messages. There are several issues, some of which we'll discuss when we talk about software for distributed- and shared-memory. However, the principal hardware issue is the cost of scaling

the interconnect. As we add processors to a bus, the chance that there will be conflicts over access to the bus increase dramatically, so buses are suitable for systems with only a few processors. Large crossbars are very expensive, so it's also unusual to find systems with large crossbar interconnects. On the other hand, distributed-memory interconnects such as the hypercube and the toroidal mesh are relatively inexpensive, and distributed-memory systems with thousands of processors that use these and other interconnects have been built. Thus, distributed-memory systems are often better suited for problems requiring vast amounts of data or computation.

Coordinating the processes/threads

In a very few cases, obtaining excellent parallel performance is trivial. For example, suppose we have two arrays and we want to add them:

```
double x[n], y[n];  
...  
for (int i = 0; i < n; i++) x[i] += y[i];
```

In order to parallelize this, we only need to assign elements of the arrays to the processes/threads. For example, if we have p processes/threads, we might make process/thread 0 responsible for elements 0, ..., $n/p - 1$, process/thread 1 would be responsible for elements n/p , ..., $2n/p - 1$, and so on.

So for this example, the programmer only needs to

Divide the work among the processes/threads

in such a way that each process/thread gets roughly the same amount of work, and

in such a way that the amount of communication required is minimized.

Recall that the process of dividing the work among the processes/threads so that

is satisfied is called **load balancing**. The two qualifications on dividing the work are obvious, but nonetheless important. In many cases it won't be necessary to give much thought to them;

they typically become concerns in situations in which the amount of work isn't known in advance by the programmer, but rather the work is generated as the program runs. For an example, see the tree search problem in Chapter 6.

Although we might wish for a term that's a little easier to pronounce, recall that the process of converting a serial program or algorithm into a parallel program is often called **parallelization**. Programs that can be parallelized by simply dividing the work among the processes/threads are sometimes said to be **embarrassingly parallel**. This is a bit unfortunate, since it suggests that programmers should be embarrassed to have written an embarrassingly parallel program, when, to the contrary, successfully devising a parallel solution to *any* problem is a cause for great rejoicing.

Alas, the vast majority of problems are much more determined to resist our efforts to find a parallel solution. As we saw in Chapter 1, for these problems, we need to coordinate the work of the processes/threads. In these programs, we also usually need to

Arrange for the processes/threads to synchronize.

Arrange for communication among the processes/threads.

These last two problems are often interrelated. For example, in distributed-memory programs, we often implicitly synchronize the processes by communicating among them, and in shared-memory programs, we often communicate among the threads by synchronizing them. We'll say more about both issues below.

Shared-memory

As we noted earlier, in shared-memory programs, variables can be **shared** or **private**. Shared variables can be read or written by any thread, and private variables can ordinarily only be accessed by one thread. Communication among the threads is usually done through shared variables, so communication is implicit, rather than explicit.

Dynamic and static threads

In many environments shared-memory programs use **dynamic threads**. In this paradigm, there is often a master thread and at any given instant a (possibly empty) collection of worker threads. The master thread typically waits for work requests— for example, over a network—and when a new request arrives, it forks a worker thread, the thread carries out the request, and when the thread completes the work, it terminates and joins the master thread. This paradigm makes efficient use of system resources since the resources required by a thread are only being used while the thread is actually running.

An alternative to the dynamic paradigm is the **static thread** paradigm. In this paradigm, all of the threads are forked after any needed setup by the master thread and the threads run until all the work is completed. After the threads join the master thread, the master thread may do some cleanup (e.g., free memory) and then it also terminates. In terms of resource usage, this may be less efficient: if a thread is idle, its resources (e.g., stack, program counter, and so on.) can't be freed. However, forking and joining threads can be fairly time-consuming operations. So if the necessary resources are available, the static thread paradigm has the potential for better performance than the dynamic paradigm. It also has the virtue that it's closer to the most widely used paradigm for distributed-memory programming, so part of the mindset that is used for one type of system is preserved for the other. Hence, we'll often use the static thread paradigm.

Nondeterminism

In any MIMD system in which the processors execute asynchronously it is likely that there will be **nondeterminism**. A computation is nondeterministic if a given input can result in different outputs. If multiple threads are executing independently, the relative rate at which they'll complete statements varies from run to run, and hence the results of the program may be different from run to run. As a very simple example, suppose we have two threads, one with id or rank 0 and the other with id or rank 1. Suppose also that each is storing a private variable my x, thread 0's value for my x is 7, and thread 1's is 19. Further, suppose both threads execute the following code:

```
...  
printf("Thread %d > my val = %d\n", my rank, my x);  
...
```

Then the output could be

Thread 0 > my_val = 7

Thread 1 > my_val = 19

but it could also be

Thread 1 > my_val = 19

Thread 0 > my_val = 7

In fact, things could be even worse: the output of one thread could be broken up by the output of the other thread. However, the point here is that because the threads are executing independently and interacting with the operating system, the time it takes for one thread to complete a block of statements varies from execution to execution, so the order in which these statements complete can't be predicted.

In many cases nondeterminism isn't a problem. In our example, since we've labelled the output with the thread's rank, the order in which the output appears probably doesn't matter. However, there are also many cases in which nondeterminism— especially in shared-memory programs—can be disastrous, because it can easily result in program errors. Here's a simple example with two threads.

Suppose each thread computes an **int**, which it stores in a private variable my_val. Suppose also that we want to add the values stored in my_val into a shared-memory location x that has been initialized to 0. Both threads therefore want to execute code that looks something like this:

```
My_val = Compute_val(my_rank);
```

```
x += my_val;
```

Now recall that an addition typically requires loading the two values to be added into registers, adding the values, and finally storing the result. To keep things relatively simple, we'll assume that values are loaded from main memory directly into registers and stored in main memory directly from registers. Here is one possible sequence of events:

| Time | Core 0 | Core 1 |
|------|-------------------------------|--------------------------------|
| 0 | Finish assignment to my_val | In call to Compute_val |
| 1 | Load x = 0 into register | Finish assignment to my_val |
| 2 | Load my_val = 7 into register | Load x = 0 into register |
| 3 | Add my_val = 7 to x | Load my_val = 19 into register |
| 4 | Store x = 7 | Add my_val to x |
| 5 | Start other work | Store x = 19 |

Clearly this is not what we want, and it's easy to imagine other sequences of events that result in an incorrect value for x. The nondeterminism here is a result of the fact that two threads are attempting to more or less simultaneously update the memory location x. When threads or processes attempt to simultaneously access a resource, and the accesses can result in an error, we often say the program has a **race condition**, because the threads or processes are in a “horse race.” That is, the out-come of the computation depends on which thread wins the race. In our example, the threads are in a race to execute `x += my_val`. In this case, unless one thread completes `x += my_val` before the other thread starts, the result will be incorrect. A block of code that can only be executed by one thread at a time is called a **critical section**, and it's usually our job as programmers to insure **mutually exclusive** access to the critical section. In other words, we need to insure that if one thread is executing the code in the critical section, then the other threads are excluded.

The most commonly used mechanism for insuring mutual exclusion is a **mutual exclusion lock** or **mutex** or **lock**. A mutex is a special type of object that has support in the underlying hardware. The basic idea is that each critical section is *protected* by a lock. Before a thread can execute the code in the critical section, it must “obtain” the mutex by calling a mutex function, and, when it's done executing the code in the critical section, it should “relinquish” the mutex by calling an unlock function. While one thread “owns” the lock—that is, has returned from a call to the lock function, but hasn't yet called the unlock function—any other thread attempting to execute the code in the critical section will wait in its call to the lock function.

Thus, in order to insure that our code functions correctly, we might modify it so that it looks something like this:

```
my_val = Compute_val(my_rank); Lock(&add_my_val_lock);
x += my_val;
```

```
Unlock(&add my_val_lock);
```

This insures that only one thread at a time can execute the statement `x += my_val`. Note that the code does *not* impose any predetermined order on the threads. Either thread 0 or thread 1 can execute `x += my_val` first.

Also note that the use of a mutex enforces **serialization** of the critical section. Since only one thread at a time can execute the code in the critical section, this code is effectively serial. Thus, we want our code to have as few critical sections as possible, and we want our critical sections to be as short as possible.

There are alternatives to mutexes. In **busy-waiting**, a thread enters a loop whose sole purpose is to test a condition. In our example, suppose there is a shared variable `ok_for_1` that has been initialized to false. Then something like the following code can insure that thread 1 won't update `x` until after thread 0 has updated it:

```
my_val = Compute_val(my_rank);
if (my_rank == 1)
    while (!ok_for_1); /* Busy-wait loop */
x += my_val;          /* Critical section */
if (my_rank == 0)
    ok_for_1 = true;  /* Let thread 1 update x */
```

```
my_val = Compute_val(my_rank);
if (my_rank == 1)
while (!ok_for_1);
/* Busy wait loop */
x += my_val; /* Critical section */
if (my_rank == 0) /* Let thread 1 update x */
ok_for_1 = true;
```

So until thread 0 executes `ok_for_1 = true`, thread 1 will be stuck in the loop **while** (`!ok_for_1`). This loop is called a “busy-wait” because the thread can be very busy waiting for the condition. This has the virtue that it's simple to understand and implement. However, it can be very wasteful of system resources, because even when a thread is doing no useful work, the core running the thread will be repeatedly checking to see if the critical section can be entered. **Semaphores** are similar to mutexes, although the details of their behavior are slightly different, and there are

some types of thread synchronization that are easier to implement with semaphores than mutexes. A **monitor** provides mutual exclusion at a somewhat higher-level: it is an object whose methods can only be executed by one thread at a time. We'll discuss busy-waiting and semaphores in Chapter 4.

There are a number of other alternatives that are currently being studied but that

are not yet widely available. The one that has attracted the most attention is probably **transactional memory** [31]. In database management systems, a **transaction** is an access to a database that the system treats as a single unit. For example, transferring \$1000 from your savings account to your checking account should be treated by your bank's software as a transaction, so that the software can't debit your savings account without also crediting your checking account. If the software was able to debit your savings account, but was then unable to credit your checking account, it would *roll-back* the transaction. In other words, the transaction would either be fully completed or any partial changes would be erased. The basic idea behind transactional memory is that critical sections in shared-memory programs should be treated as transactions. Either a thread successfully completes the critical section or any partial results are rolled back and the critical section is repeated.

Thread safety

In many, if not most, cases parallel programs can call functions developed for use in serial programs, and there won't be any problems. However, there are some notable exceptions. The most important exception for C programmers occurs in functions that make use of *static* local variables. Recall that ordinary C local variables—variables declared inside a function—are allocated from the system stack. Since each thread has its own stack, ordinary C local variables are private. However, recall that a static variable that's declared in a function persists from one call to the next. Thus, static variables are effectively shared among any threads that call the function, and this can have unexpected and unwanted consequences.

For example, the C string library function `strtok` splits an input string into sub-strings. When it's first called, it's passed a string, and on subsequent calls it returns successive substrings. This can be arranged through the use of a static **char** variable that refers to the string that was passed on the first call. Now suppose two threads are splitting strings into substrings. Clearly,

if, for example, thread 0 makes its first call to `strtok`, and then thread 1 makes its first call to `strtok` before thread 0 has completed splitting its string, then thread 0's string will be lost or overwritten, and,

on subsequent calls it may get substrings of thread 1's strings.

A function such as `strtok` is not **thread safe**. This means that if it is used in a multithreaded program, there may be errors or unexpected results. When a block of code isn't thread safe, it's usually because different threads are accessing shared data. Thus, as we've seen, even though many serial functions can be used safely in multithreaded programs—that is, they're *thread safe*—programmers need to be wary of functions that were written exclusively for use in serial programs. We'll take a closer look at thread safety in Chapters 4 and 5.

Distributed-memory

In distributed-memory programs, the cores can directly access only their own, private memories. There are several APIs that are used. However, by far the most widely used is message-passing. So we'll devote most of our attention in this section to message-passing. Then we'll take a brief look at a couple of other, less widely used, APIs.

Perhaps the first thing to note regarding distributed-memory APIs is that they can be used with shared-memory hardware. It's perfectly feasible for programmers to logically partition shared-memory into private address spaces for the various threads, and a library or compiler can implement the communication that's needed.

As we noted earlier, distributed-memory programs are usually executed by starting multiple processes rather than multiple threads. This is because typical “threads of execution” in a distributed-memory program may run on independent CPUs with independent operating systems, and there may be no software infrastructure for starting a single “distributed” process and having that process fork one or more threads on each node of the system.

Message-passing

A message-passing API provides (at a minimum) a send and a receive function. Processes

typically identify each other by ranks in the range 0, 1,, $p-1$, where p is the number of processes. So, for example, process 1 might send a message to process 0 with the following pseudo-code:

```
char message[100];  
...  
My_rank = Get_rank();  
if (my_rank == 1) {  
    sprintf(message, "Greetings from process 1");  
    Send(message, MSG_CHAR, 100, 0);  
} else if (my_rank == 0) {  
    Receive(message, MSG_CHAR, 100, 1);  
    printf("Process 0 > Received: %s\n", message);  
}
```

Here the Get rank function returns the calling process' rank. Then the processes branch depending on their ranks. Process 1 creates a message with sprintf from the standard C library and then sends it to process 0 with the call to Send. The arguments to the call are, in order, the message, the type of the elements in the message (MSG_CHAR), the number of elements in the message (100), and the rank of the destination process (0). On the other hand, process 0 calls Receive with the following arguments: the variable into which the message will be received (message), the type of the message elements, the number of elements available for storing the message, and the rank of the process sending the message. After completing the call to

Receive, process 0 prints the message.

Several points are worth noting here. First note that the program segment is SPMD. The two processes are using the same executable, but carrying out different actions. In this case, what they do depends on their ranks. Second, note that the variable message refers to different blocks of memory on the different processes. Programmers often stress this by using variable names such as my message or local message. Finally, note that we're assuming that process 0 can write to stdout. This is usually the case: most implementations of message-passing APIs allow all processes access to stdout and stderr—even if the API doesn't explicitly provide for this. We'll talk a little more about I/O later on.

There are several possibilities for the exact behavior of the Send and Receive functions, and most message-passing APIs provide several different send and/or receive functions. The

simplest behavior is for the call to Send to **block** until the call to Receive starts receiving the data. This means that the process calling Send won't return from the call until the matching call to Receive has started. Alternatively, the Send function may copy the contents of the message into storage that it owns, and then it will return as soon as the data is copied. The most common behavior for the Receive function is for the receiving process to block until the message is received. There are other possibilities for both Send and Receive, and we'll discuss some of them in Chapter 3.

Typical message-passing APIs also provide a wide variety of additional functions. For example, there may be functions for various “collective” communications, such as a **broadcast**, in which a single process transmits the same data to all the processes, or a **reduction**, in which results computed by the individual processes are combined into a single result—for example, values computed by the processes are added. There may also be special functions for managing processes and communicating complicated data structures. The most widely used API for message-passing is the **Message-Passing Interface** or MPI. We'll take a closer look at it in Chapter 3.

Message-passing is a very powerful and versatile API for developing parallel programs. Virtually all of the programs that are run on the most powerful computers in the world use message-passing. However, it is also very low level. That is, there is a huge amount of detail that the programmer needs to manage. For example, in order to parallelize a serial program, it is usually necessary to rewrite the vast majority of the program. The data structures in the program may have to either be replicated by each process or be explicitly distributed among the processes. Furthermore, the rewriting usually can't be done incrementally. For example, if a data structure is used in many parts of the program, distributing it for the parallel parts and collecting it for the serial (unparallelized) parts will probably be prohibitively expensive. Therefore, message-passing is sometimes called “the assembly language of parallel programming,” and there have been many attempts to develop other distributed-memory APIs.

One-sided communication

In message-passing, one process, must call a send function and the send must be matched by another process' call to a receive function. Any communication requires the explicit participation of two processes. In **one-sided communication**, or **remote memory access**, a single process calls a function, which updates either local memory with a value from another

process or remote memory with a value from the calling process. This can simplify communication, since it only requires the active participation of a single process. Furthermore, it can significantly reduce the cost of communication by eliminating the overhead associated with synchronizing two processes. It can also reduce overhead by eliminating the overhead of one of the function calls (send or receive).

It should be noted that some of these advantages may be hard to realize in practice. For example, if process 0 is copying a value into the memory of process 1, 0 must have some way of knowing when it's safe to copy, since it will overwrite some memory location. Process 1 must also have some way of knowing when the memory location has been updated. The first problem can be solved by synchronizing the two processes before the copy, and the second problem can be solved by another synchronization or by having a “flag” variable that process 0 sets after it has completed the copy. In the latter case, process 1 may need to **poll** the flag variable in order to determine that the new value is available. That is, it must repeatedly check the flag variable until it gets the value indicating 0 has completed its copy. Clearly, these problems can considerably increase the overhead associated with transmitting a value. A further difficulty is that since there is no explicit interaction between the two processes, remote memory operations can introduce errors that are very hard to track down.

Partitioned global address space languages

Since many programmers find shared-memory programming more appealing than message-passing or one-sided communication, a number of groups are developing parallel programming languages that allow the user to use some shared-memory techniques for programming distributed-memory hardware. This isn't quite as simple as it sounds. If we simply wrote a compiler that treated the collective memories in a distributed-memory system as a single large memory, our programs would have poor, or, at best, unpredictable performance, since each time a running process accessed memory, it might access local memory—that is, memory belonging to the core on which it was executing—or remote memory, memory belonging to another core. Accessing remote memory can take hundreds or even thousands of times longer than accessing local memory. As an example, consider the following pseudo-code for a shared-memory vector addition:

```
shared int n = . . . ;
```



```
shared double x[n], y[n];

private int i, my_first_element, my_last_element; my_first_element = . . . ;

my_last_element = . . . ;
/* Initialize x and y */
. . .
for (i = my_first_element; i <= my_last_element; i++)
x[i] += y[i];
```

We first declare two shared arrays. Then, on the basis of the process' rank, we determine which elements of the array “belong” to which process. After initializing the arrays, each process adds its assigned elements. If the assigned elements of x and y have been allocated so that the elements assigned to each process are in the memory attached to the core the process is running on, then this code should be very fast. However, if, for example, all of x is assigned to core 0 and all of y is assigned to core 1, then the performance is likely to be terrible, since each time the assignment $x[i] += y[i]$ is executed, the process will need to refer to remote memory.

Partitioned global address space, or PGAS, languages provide some of the mechanisms of shared-memory programs. However, they provide the programmer with tools to avoid the problem we just discussed. Private variables are allocated in the local memory of the core on which the process is executing, and the distribution of the data in shared data structures is controlled by the programmer. So, for example, she knows which elements of a shared array are in which process' local memory.