An Introduction to Parallel Programming

Second Edition

Peter S. Pacheco
University of San Francisco

Matthew Malensek
University of San Francisco

MORGAN KAUFMANN PUBLISHERS
AN IMPRINT OF ELSEVIER

Contents

Module 01

Module-02

Module-03

Module-03

Module-04

M-04

Module-05

Simultaneous multithreading, or SMT, is a variation on fine-grained multithreading. It attempts to exploit superscalar processors by allowing multiple threads to make use of the multiple functional units. If we designate "preferred" threads—threads that have many instructions ready to execute—we can somewhat reduce the problem of thread slowdown.

2.3 Parallel hardware

Multiple issue and pipelining can clearly be considered to be parallel hardware, since different functional units can be executed simultaneously. 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.

2.3.1 Classifications of parallel computers

We'll make use of two, independent classifications of parallel computers. The first, **Flynn's taxonomy** [20], classifies a parallel computer according to the number of instruction streams and the number of data streams (or datapaths) 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, in most cases, computes a single data value at a time. The parallel systems in the classification can always manage multiple data streams, and we differentiate between systems that support only a single instruction stream (SIMD) and systems that support multiple instruction streams (MIMD).

We already encountered the alternative classification in Chapter 1: it has to do with how the cores access memory. In **shared memory** systems, the cores can share access to memory locations, and the cores coordinate their work by modifying shared memory locations. In **distributed memory** systems, each core has its own, private memory, and the cores coordinate their work by communicating across a network.

2.3.2 SIMD systems

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 datapaths. An instruction is broadcast from the control unit to the datapaths, and each datapath 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 datapaths. Then we could load `x[i]` and `y[i]` into the ith datapath, have the ith datapath add `y[i]` to `x[i]`, and store the result in `x[i]`. If the system has m datapaths 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 datapaths will be idle.

The requirement that all the datapaths 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 a datapath and determine whether it's positive. If `y[i]` is positive, we can proceed to carry out the addition. Otherwise, the datapath storing `y[i]` will be idle while the other datapaths carry out the addition.

Note also that in a "classical" SIMD system, the datapaths must operate synchronously, that is, each datapath must wait for the next instruction to be broadcast before proceeding. Furthermore, the datapaths have no instruction storage, so a datapath 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 simple 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 one of the largest manufacturers of parallel 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 256 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" that 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. Furthermore, 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 usually 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 processing 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 datapaths (e.g., 128) 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 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 datapaths busy, and GPUs may have relatively poor performance on small problems.

It should be stressed that GPUs are not pure SIMD systems. Although the datapaths on a given core can use SIMD parallelism, current generation GPUs can run more than one instruction stream on a single core. Furthermore, typical GPUs can have dozens of cores, and these cores are also capable of executing independent instruction streams. So GPUs are neither purely SIMD nor purely MIMD.

Also note that GPUs can use shared or distributed memory. The largest systems often use both: several cores may have access to a common block of memory, but other SIMD cores may have access to a different block of shared memory, and two cores with access to different blocks of shared memory may communicate over a network. However, in the remainder of this text, we'll limit our discussion to GPUs that use shared memory.

GPUs are becoming increasingly popular for general, high-performance computing, and several languages have been developed that allow users to exploit their power. We'll go into more detail on the architecture of Nvidia processors and how to program them in Chapter 6. Also see [33].

2.3.3 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 datapath. 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 Figs. 2.3 and 2.4.

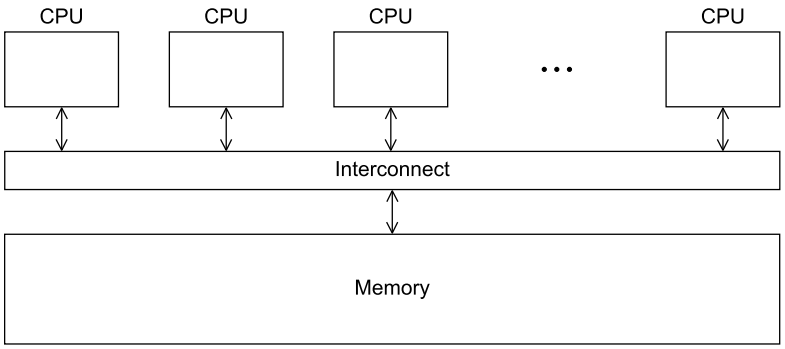

FIGURE 2.3

A shared-memory system.

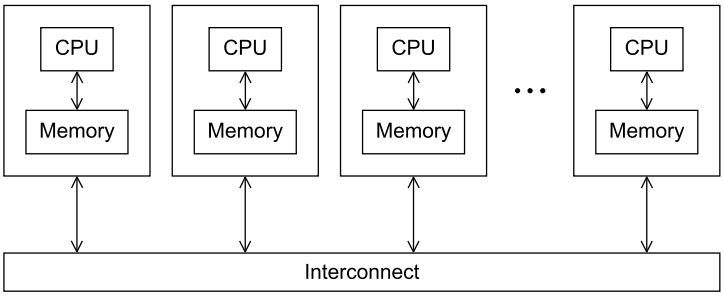

FIGURE 2.4

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.

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 other's blocks of main memory through special hardware built into the processors. (See Figs. 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

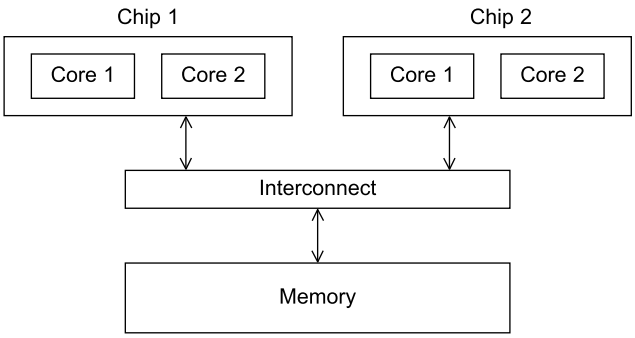

FIGURE 2.5

A UMA multicore system.

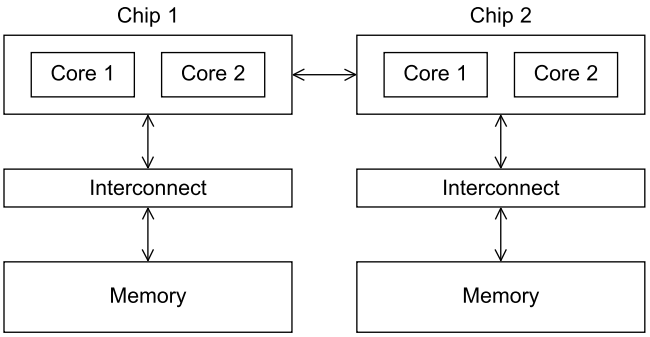

FIGURE 2.6

A NUMA multicore system.

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.

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 has 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 is *heterogeneous*, that is, the individual nodes are built from different types of hardware.

2.3.4 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 2.11.

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

In the past, it was common for shared memory systems to use a **bus** to connect processors and memory. Originally, a **bus** was 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. So, as the size of shared-memory systems has increased, buses are 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 a relatively simple and powerful switched interconnect. The diagram in Fig. 2.7(a) shows a simple crossbar. 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 Fig. 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 if the two cores attempt to simultaneously access the same memory module. For example, Fig. 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 interconnects 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. Fig. 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.

One of the simplest measures of the power of a direct interconnect is the number of links. When counting links in a direct interconnect, it's customary to count only switch-to-switch links. This is because the speed of the processor-to-switch links may be very different from the speed of the switch-to-switch links. Furthermore, to get the total number of links, we can usually just add the number of processors to the number of switch-to-switch links. So in the diagram for a ring (Fig. 2.8a), we would ordinarily count 3 links instead of 6, and in the diagram for the toroidal mesh (Fig. 2.8b), we would count 18 links instead of 27.

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 $2p$ in a toroidal mesh, while it's only p 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 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 Fig. 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

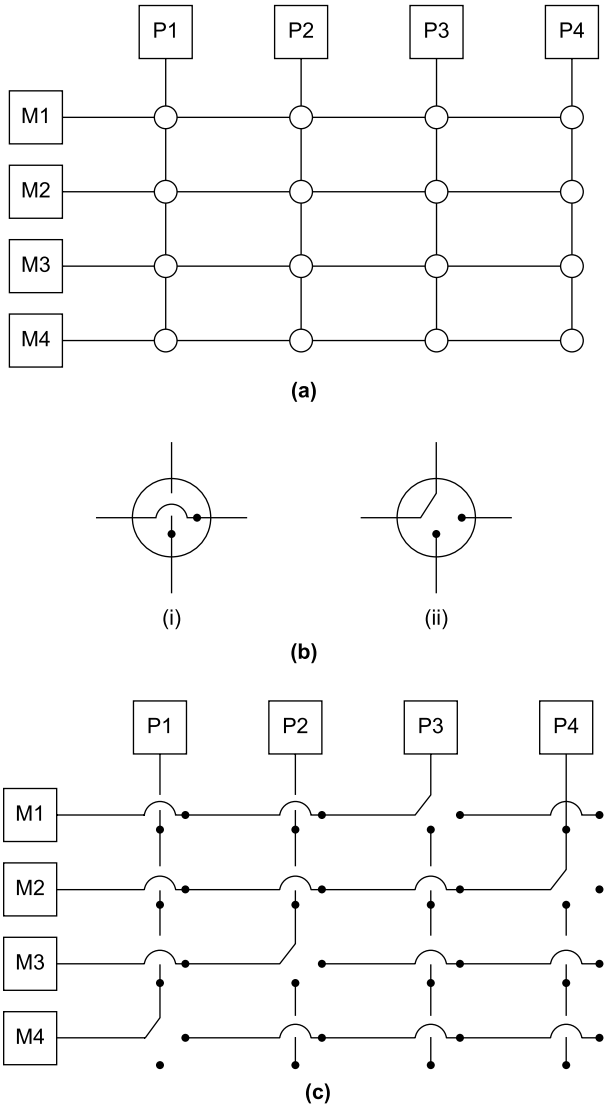

FIGURE 2.7

(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.

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 Fig. 2.9(b) we've divided the nodes into two parts so that four simultaneous communications

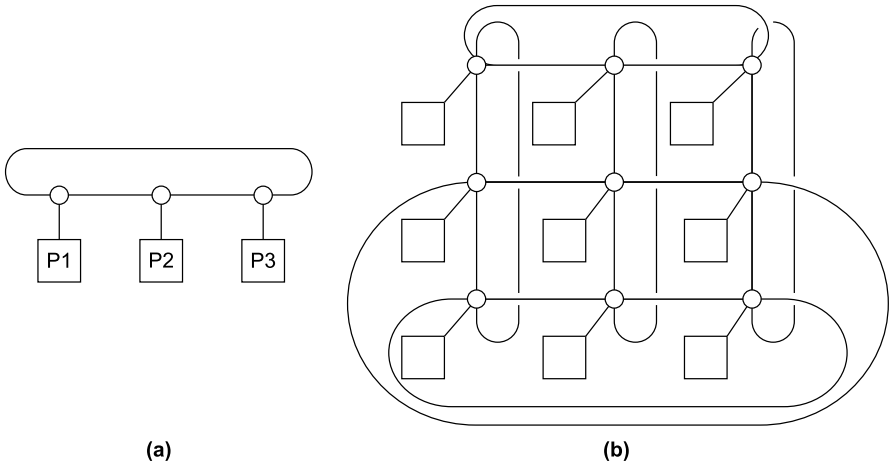

FIGURE 2.8

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

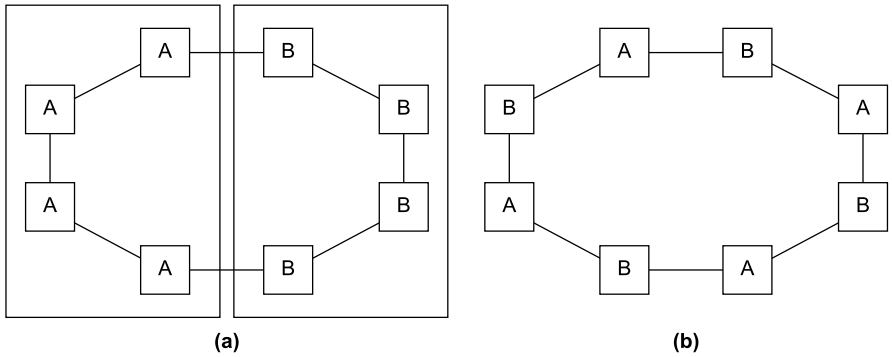

FIGURE 2.9

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

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 Fig. 2.10.) This suggests that the bisection width is at most $2q = 2\sqrt{p}$. In

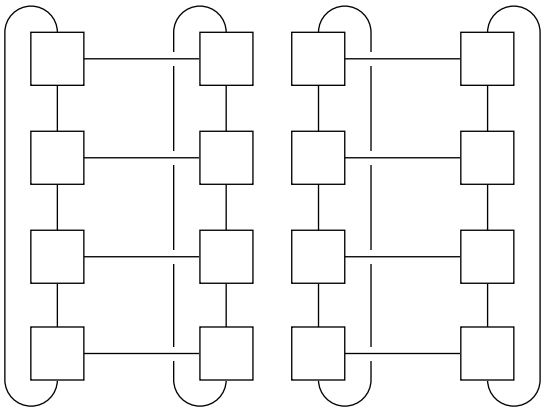

FIGURE 2.10

A bisection of a toroidal mesh.

fact, this is the smallest possible number of links, and the bisection width of a square two-dimensional toroidal mesh is $2\sqrt{p}$.

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 Fig. 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.

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 Fig. 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.

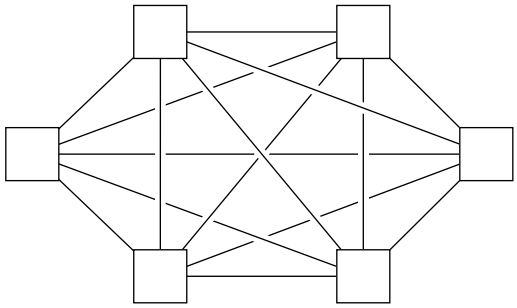

FIGURE 2.11

A fully connected network.

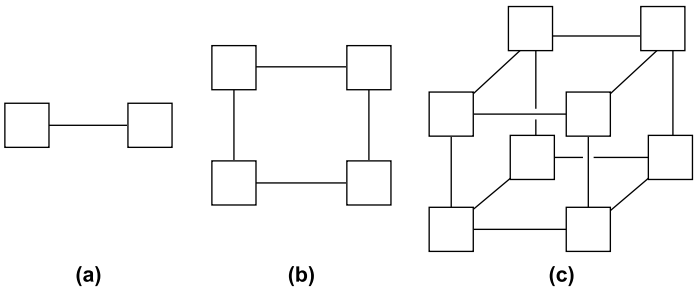

FIGURE 2.12

(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 Fig. 2.13.)

The **crossbar** and the **omega network** are relatively simple examples of indirect networks. We saw a shared-memory crossbar with bidirectional links earlier (Fig. 2.7). The diagram of a distributed-memory crossbar in Fig. 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 Fig. 2.15. The switches are two-by-two crossbars (see Fig. 2.16). Observe that unlike the crossbar, there are communications that cannot occur simultaneously. For example, in Fig. 2.15, if processor 0 sends 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.

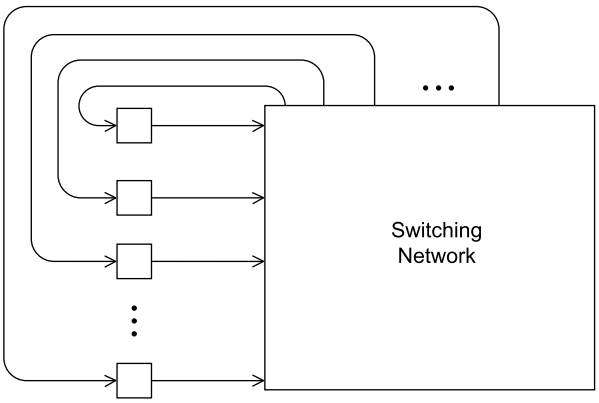

FIGURE 2.13

A generic indirect network.

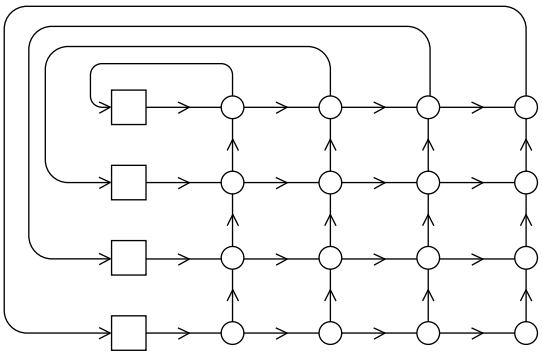

FIGURE 2.14

A crossbar interconnect for distributed-memory.

It's a little bit more complicated to define bisection width for indirect networks. 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

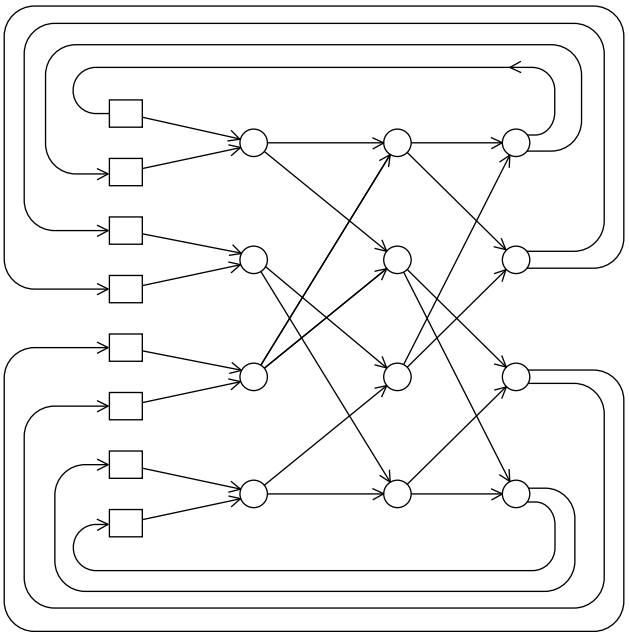

FIGURE 2.15

An omega network.

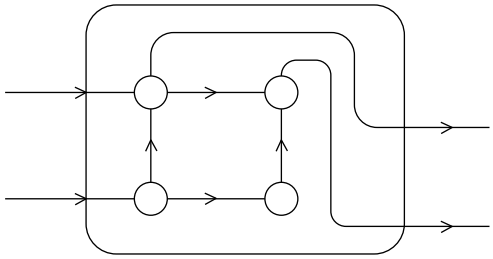

FIGURE 2.16

A switch in an omega network.

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).

2.3.5 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 Fig. 2.17.) As long as the two cores only read shared data, there is no problem. For example, suppose that

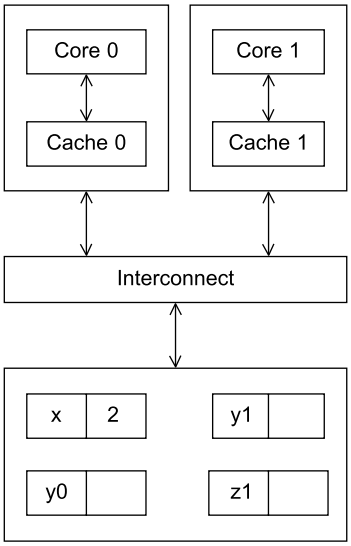

FIGURE 2.17

A shared-memory system with two cores and two caches.

x is a shared variable that has been initialized to 2, y0 is private and owned by core 0, and y1 and z1 are private and owned by core 1. Now suppose the following statements are executed at the indicated times:

Time	Core 0	Core 1
0	`y0 = x;`	`y1 = 3*x;`
1	`x = 7;`	Statement(s) not involving x
2	Statement(s) not involving x	`z1 = 4*x;`

Then the memory location for y0 will eventually get the value 2, and the memory location for y1 will eventually get the value 6. However, it's not so clear what value z1 will get. It might at first appear that since core 0 updates x to 7 before the assignment to z1, z1 will get the value $4 \times 7 = 28$. 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 z1 will get the value $4 \times 2 = 8$.

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 ensuring 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 ensuring 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. 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 (Fig. 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 will invalidate those lines.

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; 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 other's 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 main 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.

2.3.6 Shared-memory vs. 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.

2.4 Parallel software

Parallel hardware has arrived. Virtually all desktop and server systems use multicore processors. Nowadays even mobile phones and tablets make use of multicore processors. The first edition (2011) asserted, "The same cannot be said for parallel software." Currently (2021) the situation for parallel software is in flux. Most system software makes some use of parallelism, and many widely used application programs (e.g., Excel, Photoshop, Chrome) can also use multiple cores. However, there are still many programs that can only make use of a single core, and there are many programmers with no experience writing parallel programs. This is a problem, because we can no longer rely on hardware and compilers to provide a steady increase in application performance. If we're to continue to have routine increases in application performance and application power, software developers must learn to write applications that exploit shared- and distributed-memory architectures and MIMD and SIMD systems. In this section, we'll take a look at some of the issues involved in writing software for parallel systems.

First, some terminology. Typically when we run our shared-memory programs, we'll start a single process and fork multiple threads. So when we discuss shared-memory programs, we'll talk about *threads* carrying out tasks. On the other hand, when we run distributed-memory programs, we'll start multiple processes, and we'll talk about *processes* carrying out tasks. When the discussion applies equally well to shared-memory and distributed-memory systems, we'll talk about *processes/threads* carrying out tasks.

2.4.1 Caveats

Before proceeding, we need to stress some of the limitations of this section. First, we stress that our coverage in this chapter is only meant to give some idea of the issues: there is no attempt to be comprehensive.

Second, we'll mainly focus on what's often called **single program, multiple data**, or SPMD, programs. Instead of running a different program on each core, SPMD programs consist of a single executable that can behave as if it were multiple different programs through the use of conditional branches. For example,

```
if (I'm thread/process 0)
   do this;
else
   do that;
```

Observe that SPMD programs can readily implement data-parallelism. For example,

```
if (I'm thread/process 0)
   operate on the first half of the array;
else /* I'm thread/process 1 */
   operate on the second half of the array;
```

Recall that a program is **task parallel** if it obtains its parallelism by dividing tasks among the threads or processes. The first example makes it clear that SPMD programs can also implement **task-parallelism**.

2.4.2 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];
```

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 $0, \ldots, n/p - 1$, process/thread 1 would be responsible for elements $n/p, \ldots, 2n/p - 1$, and so on.

So for this example, the programmer only needs to

1. Divide the work among the processes/threads in such a way that
   a. each process/thread gets roughly the same amount of work, and
   b. the amount of communication required is minimized.

   Recall that the process of dividing the work among the processes/threads so that (a) 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.

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

2. Arrange for the processes/threads to synchronize.
3. 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.

2.4.3 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_x = %d\n", my_rank, my_x);
. . .
```

Then the output could be

```
Thread 0 > my_x = 7
Thread 1 > my_x = 19
```

but it could also be

```
Thread 1 > my_x = 19
Thread 0 > my_x = 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 labeled 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 shared 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 "race" to carry out an operation. That is, the outcome 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. So we need for each thread's operation `x += my_val` to be **atomic**. An operation that writes to a memory location is atomic if, after a thread has completed the operation, it appears that no *other* thread has modified the memory location. There are usually several ways to ensure that an operation is atomic. One possibility is to ensure that only one thread executes the update `x += my_val` at a time. 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 ensure **mutually exclusive** access to a critical section. In other words, we need to ensure that if one thread is executing the code in the critical section, then the other threads are excluded.

The most commonly used mechanism for ensuring mutual exclusion is a **mutual exclusion lock** or **mutex**, or simply **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 to ensure 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 ensures 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 ensure 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 */
```

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.

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 substrings. 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.

2.4.4 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 $0, 1, \ldots, p-1$, where p is the number of processes. So, for example, process 1 might send a message to process 0 with the following pseudocode:

```
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's 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 are 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, 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's 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 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 pseudocode 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's 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's local memory.

There are several projects currently working on the development of PGAS languages. See, for example, [8,54].

2.4.5 GPU programming

GPUs are usually not "standalone" processors. They don't ordinarily run an operating system and system services, such as direct access to secondary storage. So programming a GPU also involves writing code for the CPU "host" system, which runs on an ordinary CPU. The memory for the CPU host and the GPU memory are usually separate. So the code that runs on the host typically allocates and initializes storage on both the CPU and the GPU. It will start the program on the GPU, and it is responsible for the output of the results of the GPU program. Thus GPU programming is really *heterogeneous* programming, since it involves programming two different types of processors.

The GPU itself will have one or more processors. Each of these processors is capable of running hundreds or thousands of threads. In the systems we'll be using, the processors share a large block of memory, but each individual processor has a small block of much faster memory that can only be accessed by threads running on that processor. These blocks of faster memory can be thought of as a programmer-managed cache.

The threads running on a processor are typically divided into groups: the threads within a group use the SIMD model, and two threads in different groups can run independently. The threads in a SIMD group may not run in lockstep. That is, they may not all execute the same instruction at the same time. However, no thread in the group will execute the next instruction until all the threads in the group have completed executing the current instruction. If the threads in a group are executing a branch, it may be necessary to idle some of the threads. For example, suppose there are 32 threads in a SIMD group, and each thread has a private variable `rank_in_gp` that ranges from 0 to 31. Suppose also that the threads are executing the following code:

```
// Thread private variables
int rank_in_gp, my_x;
...
if (rank_in_gp < 16)
   my_x += 1;
else
   my_x -= 1;
```

Then the threads with rank < 16 will execute the first assignment, while the threads with rank ≥ 16 are idle. After the threads with rank < 16 are done, the roles will be reversed: the threads with rank < 16 will be idle, while the threads with rank ≥ 16 will execute the second assignment. Of course, idling half the threads for two instructions isn't a very efficient use of the available resources. So it's up to the programmer to minimize branching, where the threads within a SIMD group take different branches.

Another issue in GPU programming that's different from CPU programming is how the threads are scheduled to execute. GPUs use a hardware scheduler (unlike CPUs, which use software to schedule threads and processes), and this hardware scheduler uses very little overhead. However, the scheduler will choose to execute an instruction when all the threads in the SIMD group are ready. In the preceding example, before executing the test, we would want the variable `rank_in_gp` stored in a register by each thread. So, to maximize use of the hardware, we usually create a large number of SIMD groups. When this is the case, groups that aren't ready to execute (e.g., they're waiting for data from memory, or waiting for the completion of a previous instruction) can be idled, and the scheduler can choose a SIMD group that is ready.

2.4.6 Programming hybrid systems

Before moving on, we should note that it is possible to program systems such as clusters of multicore processors using a combination of a shared-memory API on the nodes and a distributed-memory API for internode communication. However, this is usually only done for programs that require the highest possible levels of performance, since the complexity of this "hybrid" API makes program development much more difficult. See, for example, [45]. Rather, such systems are often programmed using a single, distributed-memory API for both inter- and intra-node communication.

2.5 Input and output

2.5.1 MIMD systems

We've generally avoided the issue of input and output. There are a couple of reasons. First and foremost, parallel I/O, in which multiple cores access multiple disks or other devices, is a subject to which one could easily devote a book. See, for example, [38]. Second, the vast majority of the programs we'll develop do very little in the way of I/O. The amount of data they read and write is quite small and easily managed by the standard C I/O functions `printf`, `fprintf`, `scanf`, and `fscanf`. However, even the limited use we make of these functions can potentially cause some problems. Since these functions are part of standard C, which is a serial language, the standard says nothing about what happens when they're called by different processes. On the other hand, threads that are forked by a single process *do* share `stdin`, `stdout`, and `stderr`. However, as we've seen, when multiple threads attempt to access one of these, the outcome is nondeterministic, and it's impossible to predict what will happen.

When we call `printf` from multiple processes/threads, we, as developers, usually want the output to appear on the console of a single system, the system on which we started the program. In fact, this is what the vast majority of systems do. However, with processes, there is no guarantee, and we need to be aware that it is possible for a system to do something else: for example, only one process has access to `stdout` or `stderr`, or even *no* processes have access to `stdout` or `stderr`.

What *should* happen with calls to `scanf` when we're running multiple processes/threads is a little less obvious. Should the input be divided among the processes/threads? Or should only a single process/thread be allowed to call `scanf`? The vast majority of systems allow at least one process to call `scanf`—usually process 0—while most allow multiple threads to call `scanf`. Once again, there are some systems that don't allow any processes to call `scanf`.

When multiple processes/threads *can* access `stdout`, `stderr`, or `stdin`, as you might guess, the distribution of the input and the sequence of the output are usually nondeterministic. For output, the data will probably appear in a different order each time the program is run, or, even worse, the output of one process/thread may be broken up by the output of another process/thread. For input, the data read by each process/thread may be different on each run, even if the same input is used.

To partially address these issues, we'll be making these assumptions and following these rules when our parallel programs need to do I/O:

- In distributed-memory programs, only process 0 will access `stdin`. In shared-memory programs, only the master thread or thread 0 will access `stdin`.
- In both distributed-memory and shared-memory programs, all the processes/threads can access `stdout` and `stderr`.
- However, because of the nondeterministic order of output to `stdout`, in most cases only a single process/thread will be used for all output to `stdout`. The principal exception will be output for debugging a program. In this situation, we'll often have multiple processes/threads writing to `stdout`.
- Only a single process/thread will attempt to access any single file other than `stdin`, `stdout`, or `stderr`. So, for example, each process/thread can open its own, private file for reading or writing, but no two processes/threads will open the same file.
- Debug output should always include the rank or ID of the process/thread that's generating the output.

2.5.2 GPUs

In most cases, the host code in our GPU programs will carry out all I/O. Since we'll only be running one process/thread on the host, the standard C I/O functions should behave as they do in ordinary serial C programs.

The exception to the rule that we use the host for I/O is that when we are debugging our GPU code, we'll want to be able to write to `stdout` and/or `stderr`. In the systems we use, each thread can write to `stdout`, and, as with MIMD programs, the order of the output is nondeterministic. Also in the systems we use, *no* GPU thread has access to `stderr`, `stdin`, or secondary storage.

2.6 Performance

Of course our main purpose in writing parallel programs is usually increased performance. So what can we expect? And how can we evaluate our programs? In this section, we'll start by looking at the performance of homogeneous MIMD systems. So we'll assume that all of the cores have the same architecture. Since this is not the case for GPUs, we'll talk about the performance of GPUs in a separate subsection.

2.6.1 Speedup and efficiency in MIMD systems

Usually the best our parallel program can do is to divide the work equally among the cores while at the same time introducing no additional work for the cores. If we succeed in doing this, and we run our program with p cores, one thread or process on each core, then our parallel program will run p times faster than the serial program runs on a single core of the same design. If we call the serial run-time T_{serial} and our parallel run-time T_{parallel}, then it's usually the case that the best possible run-time

Table 2.4 Speedups and efficiencies of a parallel program.

p	1	2	4	8	16
S	1.0	1.9	3.6	6.5	10.8
$E = S/p$	1.0	0.95	0.90	0.81	0.68

of our parallel program is $T_{\text{parallel}} = T_{\text{serial}}/p$. When this happens, we say that our parallel program has **linear speedup**.

In practice, we usually don't get perfect linear speedup, because the use of multiple processes/threads almost invariably introduces some overhead. For example, shared-memory programs will almost always have critical sections, which will require that we use some mutual exclusion mechanism, such as a mutex. The calls to the mutex functions are the overhead that's not present in the serial program, and the use of the mutex forces the parallel program to serialize execution of the critical section. Distributed-memory programs will almost always need to transmit data across the network, which is usually much slower than local memory access. Serial programs, on the other hand, won't have these overheads. Thus it will be unusual for us to find that our parallel programs get linear speedup. Furthermore, it's likely that the overheads will increase as we increase the number of processes or threads. For example, more threads will probably mean more threads need to access a critical section, and more processes will probably mean more data needs to be transmitted across the network.

So if we define the **speedup** of a parallel program to be

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}},$$

then linear speedup has $S = p$. Furthermore, since as p increases we expect the parallel overhead to increase, we also expect S to become a smaller and smaller fraction of the ideal, linear speedup p. Another way of saying this is that S/p will probably get smaller and smaller as p increases. Table 2.4 shows an example of the changes in S and S/p as p increases.[1] This value, S/p, is sometimes called the **efficiency** of the parallel program. If we substitute the formula for S, we see that the efficiency is

$$\begin{aligned} E &= \frac{S}{p} = \frac{\left(\frac{T_{\text{serial}}}{T_{\text{parallel}}}\right)}{p} \\ &= \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}}. \end{aligned}$$

If the serial run-time has been taken on the same type of core that the parallel system is using, we can think of efficiency as the average utilization of the parallel

[1] These data are taken from Chapter 3. See Tables 3.6 and 3.7.

Table 2.5 Speedups and efficiencies of parallel program on different problem sizes.

	p	1	2	4	8	16
Half	S	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original	S	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double	S	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89

cores on solving the problem. That is, the efficiency can be thought of as the fraction of the parallel run-time that's spent, on average, by each core working on solving the original problem. The remainder of the parallel run-time is the parallel overhead. This can be seen by simply multiplying the efficiency and the parallel run-time:

$$E \cdot T_{\text{parallel}} = \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}} \cdot T_{\text{parallel}} = \frac{T_{\text{serial}}}{p}.$$

For example, suppose we have $T_{\text{serial}} = 24$ ms, $p = 8$, and $T_{\text{parallel}} = 4$ ms. Then

$$E = \frac{24}{8 \cdot 4} = \frac{3}{4},$$

and, on average, each process/thread spends $3/4 \cdot 4 = 3$ ms on solving the original problem, and $4 - 3 = 1$ ms in parallel overhead.

Many parallel programs are developed by explicitly dividing the work of the serial program among the processes/threads and adding in the necessary "parallel overhead," such as mutual exclusion or communication. Therefore if T_{overhead} denotes this parallel overhead, it's often the case that

$$T_{\text{parallel}} = T_{\text{serial}}/p + T_{\text{overhead}}.$$

When this formula applies, it's clear that the parallel efficiency is just the fraction of the parallel run-time spent by the parallel program in the original problem, because the formula divides the parallel run-time into the time on the original problem, T_{serial}/p, and the time spent in parallel overhead, T_{overhead}.

We've already seen that T_{parallel}, S, and E depend on p, the number of processes or threads. We also need to keep in mind that T_{parallel}, S, E, and T_{serial} all depend on the problem size. For example, if we halve and double the problem size of the program, whose speedups are shown in Table 2.4, we get the speedups and efficiencies shown in Table 2.5. The speedups are plotted in Fig. 2.18, and the efficiencies are plotted in Fig. 2.19.

We see that in this example, when we increase the problem size, the speedups and the efficiencies increase, while they decrease when we decrease the problem size.

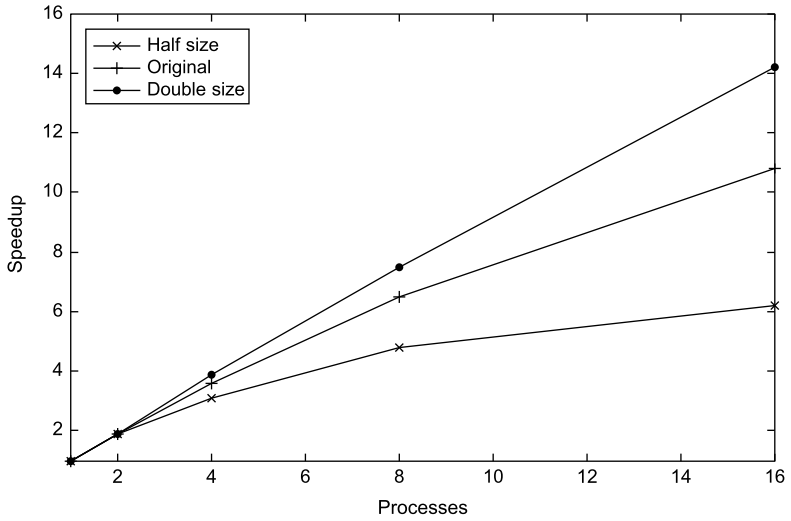

FIGURE 2.18

Speedups of parallel program on different problem sizes.

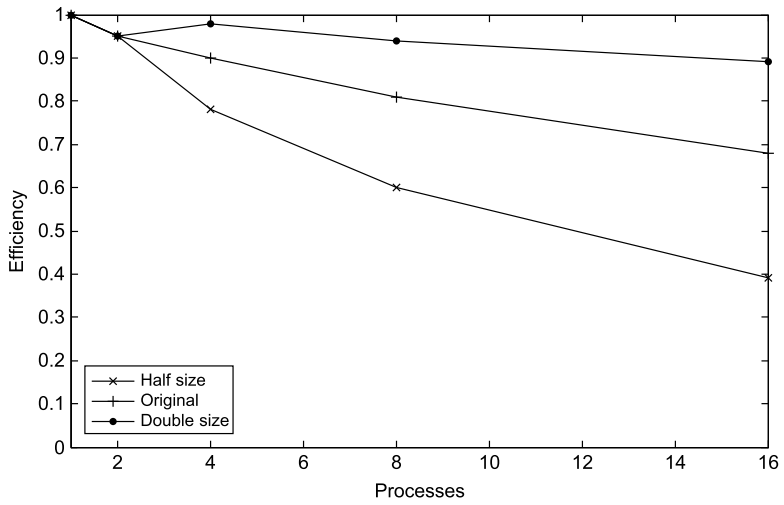

FIGURE 2.19

Efficiencies of parallel program on different problem sizes.

This behavior is quite common, because in many parallel programs, as the problem size is increased but the number of processes/threads is fixed, the parallel overhead grows much more slowly than the time spent in solving the original problem. That is,

if we think of T_{serial} and T_{overhead} as functions of the problem size, T_{serial} grows much faster as the problem size is increased. Exercise 2.15 goes into more detail.

A final issue to consider is what values of T_{serial} should be used when reporting speedups and efficiencies. Some authors say that T_{serial} should be the run-time of the fastest program on the fastest processor available. However, since it's often the case that we think of efficiency as the utilization of the cores on the parallel system, in practice, most authors use a serial program, on which the parallel program was based and run it on a single processor of the parallel system. So if we were studying the performance of a parallel shell sort program, authors in the first group might use a serial radix sort or quicksort on a single core of the fastest system available, while authors in the second group would use a serial shell sort on a single processor of the parallel system. We'll generally use the second approach.

2.6.2 Amdahl's law

Back in the 1960s, Gene Amdahl made an observation [3] that's become known as **Amdahl's law**. It says, roughly, that unless virtually all of a serial program is parallelized, the possible speedup is going to be very limited—regardless of the number of cores available. Suppose, for example, that we're able to parallelize 90% of a serial program. Furthermore, suppose that the parallelization is "perfect," that is, regardless of the number of cores p we use, the speedup of this part of the program will be p. If the serial run-time is $T_{\text{serial}} = 20$ seconds, then the run-time of the parallelized part will be $0.9 \times T_{\text{serial}}/p = 18/p$ and the run-time of the "unparallelized" part will be $0.1 \times T_{\text{serial}} = 2$. The overall parallel run-time will be

$$T_{\text{parallel}} = 0.9 \times T_{\text{serial}}/p + 0.1 \times T_{\text{serial}} = 18/p + 2,$$

and the speedup will be

$$S = \frac{T_{\text{serial}}}{0.9 \times T_{\text{serial}}/p + 0.1 \times T_{\text{serial}}} = \frac{20}{18/p + 2}.$$

Now as p gets larger and larger, $0.9 \times T_{\text{serial}}/p = 18/p$ gets closer and closer to 0, so the total parallel run-time can't be smaller than $0.1 \times T_{\text{serial}} = 2$. That is, the denominator in S can't be smaller than $0.1 \times T_{\text{serial}} = 2$. The fraction S must therefore satisfy the inequality

$$S \leq \frac{T_{\text{serial}}}{0.1 \times T_{\text{serial}}} = \frac{20}{2} = 10.$$

That is, $S \leq 10$. This is saying that even though we've done a perfect job in parallelizing 90% of the program, and even if we have, say, 1000 cores, we'll never get a speedup better than 10.

More generally, if a fraction r of our serial program remains unparallelized, then Amdahl's law says we can't get a speedup better than $1/r$. In our example, $r = 1 - 0.9 = 1/10$, so we couldn't get a speedup better than 10. Therefore if a

fraction r of our serial program is "inherently serial," that is, cannot possibly be parallelized, then we can't possibly get a speedup better than $1/r$. Thus even if r is quite small—say, 1/100—and we have a system with thousands of cores, we can't possibly get a speedup better than 100.

This is pretty daunting. Should we give up and go home? Well, no. There are several reasons not to be too worried by Amdahl's law. First, it doesn't take into consideration the problem size. For many problems, as we increase the problem size, the "inherently serial" fraction of the program decreases in size; a more mathematical version of this statement is known as **Gustafson's law** [25]. Second, there are thousands of programs used by scientists and engineers that routinely obtain huge speedups on large distributed-memory systems. Finally, is a small speedup so awful? In many cases, obtaining a speedup of 5 or 10 is more than adequate, especially if the effort involved in developing the parallel program wasn't very large.

2.6.3 Scalability in MIMD systems

The word "scalable" has a wide variety of informal uses. Indeed, we've used it several times already. Roughly speaking, a program is scalable if, by increasing the power of the system it's run on (e.g., increasing the number of cores), we can obtain speedups over the program when it's run on a less powerful system (e.g., a system with fewer cores). However, in discussions of MIMD parallel program performance, scalability has a somewhat more formal definition. Suppose we run a parallel program with a fixed number of processes/threads and a fixed input size, and we obtain an efficiency E. Suppose we now increase the number of processes/threads that are used by the program. If we can find a corresponding rate of increase in the problem size so that the program always has efficiency E, then the program is **scalable**.

As an example, suppose that $T_{\text{serial}} = n$, where the units of T_{serial} are in microseconds, and n is also the problem size. Also suppose that $T_{\text{parallel}} = n/p + 1$. Then

$$E = \frac{n}{p(n/p + 1)} = \frac{n}{n + p}.$$

To see if the program is scalable, we increase the number of processes/threads by a factor of k, and we want to find the factor x that we need to increase the problem size by, so that E is unchanged. The number of processes/threads will be kp; the problem size will be xn, and we want to solve the following equation for x:

$$E = \frac{n}{n + p} = \frac{xn}{xn + kp}.$$

Well, if $x = k$, there will be a common factor of k in the denominator $xn + kp = kn + kp = k(n + p)$, and we can reduce the fraction to get

$$\frac{xn}{xn + kp} = \frac{kn}{k(n + p)} = \frac{n}{n + p}.$$

In other words, if we increase the problem size at the same rate that we increase the number of processes/threads, then the efficiency will be unchanged, and our program is scalable.

There are a couple of cases that have special names. If, when we increase the number of processes/threads, we can keep the efficiency fixed *without* increasing the problem size, the program is said to be *strongly scalable*. If we can keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes/threads, then the program is said to be *weakly scalable*. The program in our example would be weakly scalable.

2.6.4 Taking timings of MIMD programs

You may have been wondering how we find T_{serial} and T_{parallel}. There are a *lot* of different approaches, and with parallel programs the details may depend on the API. However, there are a few general observations we can make that may make things a little easier.

The first thing to note is that there are at least two different reasons for taking timings. During program development, we may take timings to determine if the program is behaving as we intend. For example, in a distributed-memory program, we might be interested in finding out how much time the processes are spending waiting for messages, because if this value is large, there is almost certainly something wrong either with our design or our implementation. On the other hand, once we've completed development of the program, we're often interested in determining how good its performance is. Perhaps surprisingly, the way we take these two timings is usually different. For the first timing, we usually need very detailed information: How much time did the program spend in this part of the program? How much time did it spend in that part? For the second, we usually report a single value. Right now we'll talk about the second type of timing. See Exercise 2.21 for a brief discussion of one issue in taking the first type of timing.

Second, we're usually *not* interested in the time that elapses between the program's start and the program's finish. We're usually interested only in some part of the program. For example, if we write a program that implements bubble sort, we're probably only interested in the time it takes to sort the keys, not the time it takes to read them in and print them out. So we probably can't use something like the Unix shell command `time`, which reports the time taken to run a program from start to finish.

Third, we're usually *not* interested in "CPU time." This is the time reported by the standard C function `clock`. It's the total time the program spends in code executed as part of the program. It would include the time for code we've written; it would include the time we spend in library functions, such as `pow` or `sin`; and it would include the time the operating system spends in functions we call, such as `printf` and `scanf`. It would not include time the program was idle, and this could be a problem. For example, in a distributed-memory program, a process that calls a receive function may have to wait for the sending process to execute the matching send, and the operating system might put the receiving process to sleep while it waits. This idle time

wouldn't be counted as CPU time, since no function that's been called by the process is active. However, it should count in our evaluation of the overall run-time, since it may be a real cost in our program. If each time the program is run, the process has to wait, ignoring the time it spends waiting would give a misleading picture of the actual run-time of the program.

Thus when you see an article reporting the run-time of a parallel program, the reported time is usually "wall clock" time. That is, the authors of the article report the time that has elapsed between the start and finish of execution of the code that the user is interested in. If the user could see the execution of the program, she would hit the start button on her stopwatch when it begins execution and hit the stop button when it stops execution. Of course, she can't see her code executing, but she can modify the source code so that it looks something like this:

```
double start, finish;
. . .
start = Get_current_time();
/* Code that we want to time */
. . .
finish = Get_current_time();
printf("The elapsed time = %e seconds\n", finish-start);
```

The function `Get_current_time()` is a hypothetical function that's supposed to return the number of seconds that have elapsed since some fixed time in the past. It's just a placeholder. The actual function that is used will depend on the API. For example, MPI has a function `MPI_Wtime` that could be used here, and the OpenMP API for shared-memory programming has a function `omp_get_wtime`. Both functions return wall clock time instead of CPU time.

There may be an issue with the **resolution** of the timer function. The resolution is the unit of measurement on the timer. It's the duration of the shortest event that can have a nonzero time. Some timer functions have resolutions in milliseconds (10^{-3} seconds), and when instructions can take times that are less than a nanosecond (10^{-9} seconds), a program may have to execute millions of instructions before the timer reports a nonzero time. Many APIs provide a function that reports the resolution of the timer. Other APIs specify that a timer must have a given resolution. In either case we, as the programmers, need to check these values.

When we're timing parallel programs, we need to be a little more careful about how the timings are taken. In our example, the code that we want to time is probably being executed by multiple processes or threads, and our original timing will result in the output of p elapsed times:

```
private double start, finish;
. . .
start = Get_current_time();
/* Code that we want to time */
. . .
finish = Get_current_time();
printf("The elapsed time = %e seconds\n", finish-start);
```

However, what we're usually interested in is a single time: the time that has elapsed from when the first process/thread began execution of the code to the time the last process/thread finished execution of the code. We often can't obtain this exactly, since there may not be any correspondence between the clock on one node and the clock on another node. We usually settle for a compromise that looks something like this:

```
shared double global_elapsed;
private double my_start, my_finish, my_elapsed;
. . .
/* Synchronize all processes/threads */
Barrier();
my_start = Get_current_time();

/* Code that we want to time */
. . .

my_finish = Get_current_time();
my_elapsed = my_finish - my_start;

/* Find the max across all processes/threads */
global_elapsed = Global_max(my_elapsed);
if (my_rank == 0)
   printf("The elapsed time = %e seconds\n",
          global_elapsed);
```

Here, we first execute a **barrier** function that approximately synchronizes all of the processes/threads. We would like for all the processes/threads to return from the call simultaneously, but such a function usually can only guarantee that all the processes/threads have started the call when the first process/thread returns. We then execute the code as before, and each process/thread finds the time it took. Then all the processes/threads call a global maximum function, which returns the largest of the elapsed times, and process/thread 0 prints it out.

We also need to be aware of the *variability* in timings. When we run a program several times, it's extremely likely that the elapsed time will be different for each run. This will be true, even if each time we run the program we use the same input and the same systems. It might seem that the best way to deal with this would be to report either a mean or a median run-time. However, it's unlikely that some outside event could actually make our program run faster than its best possible run-time. So instead of reporting the mean or median time, we usually report the *minimum* time.

Running more than one thread per core can cause dramatic increases in the variability of timings. More importantly, if we run more than one thread per core, the system will have to take extra time to schedule and deschedule cores, and this will add to the overall run-time. Therefore we rarely run more than one thread per core.

Finally, as a practical matter, since our programs won't be designed for high-performance I/O, we'll usually not include I/O in our reported run-times.

2.6.5 GPU performance

In our discussion of MIMD performance, there is an assumption that we can evaluate a parallel program by comparing its performance to the performance of a serial program. We can, of course, compare the performance of a GPU program to the performance of a serial program, and it's quite common to see reported *speedups* of GPU programs over serial programs or parallel MIMD programs.

However, as we noted in our discussion of efficiency of MIMD programs, we often assume that the serial program is run on the same type of core that's used by the parallel computer. Since GPUs use cores that are inherently parallel, this type of comparison usually doesn't make sense for GPUs. So efficiency is ordinarily not used in discussions of the performance of GPUs.

Similarly, since the cores on the GPU are fundamentally different from conventional CPUs, it doesn't make sense to talk about linear speedup of a GPU program relative to a serial CPU program.

Note that since efficiency of a GPU program relative to a CPU program doesn't make sense, the formal definition of the scalability of a MIMD program can't be applied to a GPU program. However, the informal usage of scalability is routinely applied to GPUs: a GPU program is scalable if we can increase the size of the GPU and obtain speedups over the performance of the program on a smaller GPU.

If we run the inherently serial part of a GPU program on a conventional, serial processor, then Amdahl's law can be applied to GPU programs, and the resulting upper bound on the possible speedup will be the same as the upper bound on the possible speedup for a MIMD program. That is, if a fraction r of the original serial program isn't parallelized, and this fraction is run on a conventional serial processor, then the best possible speedup of the program running on the GPU and the serial processor will be less than $1/r$.

It should be noted that the same caveats that apply to Amdahl's law on MIMD systems also apply to Amdahl's law on GPUs: It's likely that the "inherently serial" fraction will depend on the problem size, and if it gets smaller as the problem size increases, the bound on the best possible speedup will increase. Also, *many* GPU programs obtain *huge* speedups, and, finally, a relatively small speedup may be perfectly adequate.

The same basic ideas about timing that we discussed for MIMD programs also apply to GPU programs. However, since a GPU program is ordinarily started and finished on a conventional CPU, as long as we're interested in the performance of the entirety of the program running on the GPU, we can usually just use the timer on the CPU, starting it before the GPU part(s) of the program are started, and stopping it after the GPU part(s) are done. There are more complicated scenarios—e.g., running a program on multiple CPU-GPU pairs—that require more care, but we won't be dealing with these types of programs. If we only want to time a subset of the code running on the GPU, we'll need to use a timer defined by the API for the GPU.

CHAPTER

Distributed memory programming with MPI

3

Recall that the world of parallel multiple instruction, multiple data, or MIMD, computers is, for the most part, divided into **distributed-memory** and **shared-memory** systems. From a programmer's point of view, a distributed-memory system consists of a collection of core-memory pairs connected by a network, and the memory associated with a core is directly accessible only to that core. (See Fig. 3.1.) On the other hand, from a programmer's point of view, a shared-memory system consists of a collection of cores connected to a globally accessible memory, in which each core can have access to any memory location. (See Fig. 3.2.) In this chapter, we're going to start looking at how to program distributed memory systems using **message-passing**.

Recall that in message-passing programs, a program running on one core-memory pair is usually called a **process**, and two processes can communicate by calling functions: one process calls a *send* function and the other calls a *receive* function. The implementation of message-passing that we'll be using is called **MPI**, which is an abbreviation of **Message-Passing Interface**. MPI is not a new programming language. It defines a *library* of functions that can be called from C and Fortran programs. We'll learn about some of MPI's different send and receive functions. We'll also

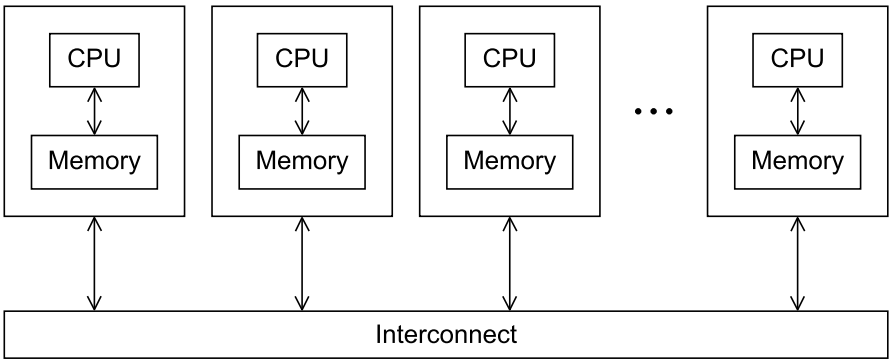

FIGURE 3.1

A distributed memory system.

An Introduction to Parallel Programming. https://doi.org/10.1016/B978-0-12-804605-0.00010-5

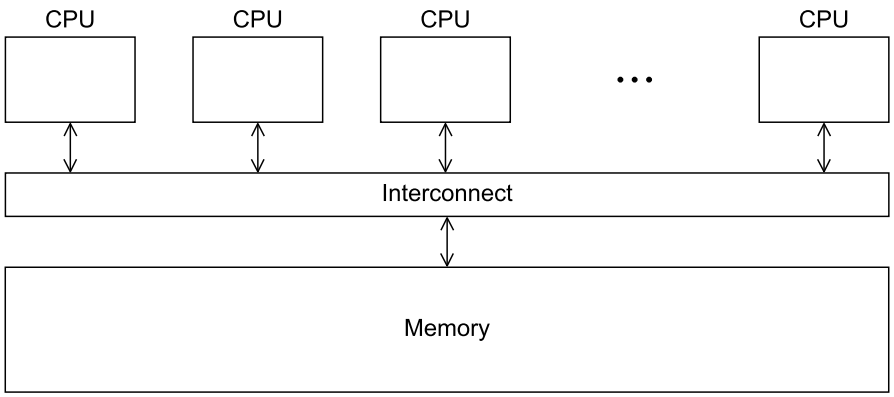

FIGURE 3.2

A shared memory system.

learn about some "global" communication functions that can involve more than two processes. These functions are called **collective** communications. In the process of learning about all of these MPI functions, we'll also learn about some of the fundamental issues involved in writing message-passing programs—issues such as data partitioning and I/O in distributed-memory systems. We'll also revisit the issue of parallel program performance.

3.1 Getting started

Perhaps the first program that many of us saw was some variant of the "hello, world" program in Kernighan and Ritchie's classic text [32]:

```
#include <stdio.h>

int main(void) {
   printf("hello, world\n");

   return 0;
}
```

Let's write a program similar to "hello, world" that makes some use of MPI. Instead of having each process simply print a message, we'll designate one process to do the output, and the other processes will send it messages, which it will print.

In parallel programming, it's common (one might say standard) for the processes to be identified by nonnegative integer *ranks*. So if there are p processes, the pro-

cesses will have ranks $0, 1, 2, \ldots, p-1$. For our parallel "hello, world," let's make process 0 the designated process, and the other processes will send it messages. See Program 3.1.

```
1  #include <stdio.h>
2  #include <string.h>  /* For strlen             */
3  #include <mpi.h>     /* For MPI functions, etc */
4
5  const int MAX_STRING = 100;
6
7  int main(void) {
8     char        greeting[MAX_STRING];
9     int         comm_sz;  /* Number of processes */
10    int         my_rank;  /* My process rank     */
11
12    MPI_Init(NULL, NULL);
13    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
14    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
15
16    if (my_rank != 0) {
17       sprintf(greeting, "Greetings from process %d of %d!",
18             my_rank, comm_sz);
19       MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
20             MPI_COMM_WORLD);
21    } else {
22       printf("Greetings from process %d of %d!\n",
23             my_rank, comm_sz);
24       for (int q = 1; q < comm_sz; q++) {
25          MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
26                0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
27          printf("%s\n", greeting);
28       }
29    }
30
31    MPI_Finalize();
32    return 0;
33 }  /* main */
```

Program 3.1: MPI program that prints greetings from the processes.

3.1.1 Compilation and execution

The details of compiling and running the program depend on your system, so you may need to check with a local expert. However, recall that when we need to be explicit, we'll assume that we're using a text editor to write the program source, and

the command line to compile and run. Many systems use a command called `mpicc` for compilation[1]:

```
$ mpicc -g -Wall -o mpi_hello mpi_hello.c
```

Typically `mpicc` is a script that's a **wrapper** for the C compiler. A **wrapper script** is a script whose main purpose is to run some program. In this case, the program is the C compiler. However, the wrapper simplifies the running of the compiler by telling it where to find the necessary header files, and which libraries to link with the object file.

Many systems also support program startup with `mpiexec`:

```
$ mpiexec -n <number of processes> ./mpi_hello
```

So to run the program with one process, we'd type

```
$ mpiexec -n 1 ./mpi_hello
```

and to run the program with four processes, we'd type

```
$ mpiexec -n 4 ./mpi_hello
```

With one process, the program's output would be

```
Greetings from process 0 of 1!
```

and with four processes, the program's output would be

```
Greetings from process 0 of 4!
Greetings from process 1 of 4!
Greetings from process 2 of 4!
Greetings from process 3 of 4!
```

How do we get from invoking `mpiexec` to one or more lines of greetings? The `mpiexec` command tells the system to start <`number of processes`> instances of our <`mpi_hello`> program. It may also tell the system which core should run each instance of the program. After the processes are running, the MPI implementation takes care of making sure that the processes can communicate with each other.

3.1.2 MPI programs

Let's take a closer look at the program.

The first thing to observe is that this *is* a C program. For example, it includes the standard C header files `stdio.h` and `string.h`. It also has a `main` function, just like any other C program. However, there are many parts of the program that are new. Line 3 includes the `mpi.h` header file. This contains prototypes of MPI functions, macro definitions, type definitions, and so on; it contains all the definitions and declarations needed for compiling an MPI program.

[1] Recall that the dollar sign (`$`) is the shell prompt—so it shouldn't be typed in. Also recall that, for the sake of explicitness, we assume that we're using the Gnu C compiler, `gcc`, and we always use the options `-g, -Wall, and -o`. See Section 2.9 for further information.

The second thing to observe is that all of the identifiers defined by MPI start with the string `MPI_`. The first letter following the underscore is capitalized for function names and MPI-defined types. All of the letters in MPI-defined macros and constants are capitalized. So there's no question about what is defined by MPI and what's defined by the user program.

3.1.3 `MPI_Init` and `MPI_Finalize`

In Line 12, the call to `MPI_Init` tells the MPI system to do all of the necessary setup. For example, it might allocate storage for message buffers, and it might decide which process gets which rank. As a rule of thumb, no other MPI functions should be called before the program calls `MPI_Init`. Its syntax is

```
int MPI_Init(
      int*      argc_p  /* in/out */,
      char***   argv_p  /* in/out */);
```

The arguments, `argc_p` and `argv_p`, are pointers to the arguments to `main`, `argc` and `argv`. However, when our program doesn't use these arguments, we can just pass `NULL` for both. Like most MPI functions, `MPI_Init` returns an **int** error code. In most cases, we'll ignore the error codes, since checking them tends to clutter the code and make it more difficult to understand what it's doing.[2]

In Line 31, the call to `MPI_Finalize` tells the MPI system that we're done using MPI, and that any resources allocated for MPI can be freed. The syntax is quite simple:

```
int MPI_Finalize(void);
```

In general, no MPI functions should be called after the call to `MPI_Finalize`.

Thus a typical MPI program has the following basic outline:

```
. . .
#include <mpi.h>
. . .
int main(int argc, char* argv[]) {
   . . .
   /* No MPI calls before this */
   MPI_Init(&argc, &argv);
   . . .
   MPI_Finalize();
   /* No MPI calls after this */
   . . .
   return 0;
}
```

[2] Of course, when we're debugging our programs, we may make extensive use of the MPI error codes.

However, we've already seen that it's not necessary to pass pointers to `argc` and `argv` to `MPI_Init`. It's also not necessary that the calls to `MPI_Init` and `MPI_Finalize` be in `main`.

3.1.4 Communicators, `MPI_Comm_size`, and `MPI_Comm_rank`

In MPI a **communicator** is a collection of processes that can send messages to each other. One of the purposes of `MPI_Init` is to define a communicator that consists of all of the processes started by the user when she started the program. This communicator is called `MPI_COMM_WORLD`. The function calls in Lines 13 and 14 are getting information about `MPI_COMM_WORLD`. Their syntax is

```
int MPI_Comm_size(
      MPI_Comm   comm          /* in  */,
      int*       comm_sz_p     /* out */);

int MPI_Comm_rank(
      MPI_Comm   comm          /* in  */,
      int*       my_rank_p     /* out */);
```

For both functions, the first argument is a communicator and has the special type defined by MPI for communicators, `MPI_Comm`. `MPI_Comm_size` returns in its second argument the number of processes in the communicator, `MPI_Comm_rank` returns in its second argument the calling process's rank in the communicator. We'll often use the variable `comm_sz` for the number of processes in `MPI_COMM_WORLD`, and the variable `my_rank` for the process rank.

3.1.5 SPMD programs

Notice that we compiled a single program—we didn't compile a different program for each process—and we did this in spite of the fact that process 0 is doing something fundamentally different from the other processes: it's receiving a series of messages and printing them, while each of the other processes is creating and sending a message. This is quite common in parallel programming. In fact, *most* MPI programs are written in this way. That is, a single program is written so that different processes carry out different actions, and this can be achieved by simply having the processes branch on the basis of their process rank. Recall that this approach to parallel programming is called **single program, multiple data** or **SPMD**. The **if**–**else** statement in Lines 16–29 makes our program SPMD.

Also notice that our program will, in principle, run with any number of processes. We saw a little while ago that it can be run with one process or four processes, but if our system has sufficient resources, we could also run it with 1000 or even 100,000 processes. Although MPI doesn't require that programs have this property, it's almost always the case that we try to write programs that will run with any number of processes, because we usually don't know in advance the exact resources available to us. For example, we might have a 20-core system available today, but tomorrow we might have access to a 500-core system.

3.1.6 Communication

In Lines 17–18, each process, other than process 0, creates a message it will send to process 0. (The function `sprintf` is very similar to `printf`, except that instead of writing to `stdout`, it writes to a string.) Lines 19–20 actually send the message to process 0. Process 0, on the other hand, simply prints its message using `printf`, and then uses a **for** loop to receive and print the messages sent by processes $1, 2, \ldots,$ `comm_sz` $- 1$. Lines 25–26 receive the message sent by process q, for $q = 1, 2, \ldots,$ `comm_sz` $- 1$.

3.1.7 MPI_Send

The sends executed by processes $1, 2, \ldots,$ `comm_sz` $- 1$ are fairly complex, so let's take a closer look at them. Each of the sends is carried out by a call to `MPI_Send`, whose syntax is

```
int MPI_Send(
      void*          msg_buf_p      /* in */,
      int            msg_size       /* in */,
      MPI_Datatype   msg_type       /* in */,
      int            dest           /* in */,
      int            tag            /* in */,
      MPI_Comm       communicator   /* in */);
```

The first three arguments, `msg_buf_p`, `msg_size`, and `msg_type`, determine the contents of the message. The remaining arguments, `dest`, `tag`, and `communicator`, determine the destination of the message.

The first argument, `msg_buf_p`, is a pointer to the block of memory containing the contents of the message. In our program, this is just the string containing the message, `greeting`. (Remember that in C an array, such as a string, is a pointer.) The second and third arguments, `msg_size` and `msg_type`, determine the amount of data to be sent. In our program, the `msg_size` argument is the number of characters in the message, plus one character for the `'\0'` character that terminates C strings. The `msg_type` argument is `MPI_CHAR`. These two arguments together tell the system that the message contains `strlen(greeting)`+1 **char**s.

Since C types (**int**, **char**, etc.) can't be passed as arguments to functions, MPI defines a special type, `MPI_Datatype`, that is used for the `msg_type` argument. MPI also defines a number of constant values for this type. The ones we'll use (and a few others) are listed in Table 3.1.

Notice that the size of the string `greeting` is not the same as the size of the message specified by the arguments `msg_size` and `msg_type`. For example, when we run the program with four processes, the length of each of the messages is 31 characters, while we've allocated storage for 100 characters in `greetings`. Of course, the size of the message sent should be less than or equal to the amount of storage in the buffer—in our case the string `greeting`.

The fourth argument, `dest`, specifies the rank of the process that should receive the message. The fifth argument, `tag`, is a nonnegative **int**. It can be used to distinguish

Table 3.1 Some predefined MPI datatypes.

MPI datatype	C datatype
MPI_CHAR	signed **char**
MPI_SHORT	signed **short int**
MPI_INT	signed **int**
MPI_LONG	signed **long int**
MPI_LONG_LONG	signed **long long int**
MPI_UNSIGNED_CHAR	**unsigned char**
MPI_UNSIGNED_SHORT	**unsigned short int**
MPI_UNSIGNED	**unsigned int**
MPI_UNSIGNED_LONG	**unsigned long int**
MPI_FLOAT	**float**
MPI_DOUBLE	**double**
MPI_LONG_DOUBLE	**long double**
MPI_BYTE	
MPI_PACKED	

messages that are otherwise identical. For example, suppose process 1 is sending **float**s to process 0. Some of the **float**s should be printed, while others should be used in a computation. Then the first four arguments to `MPI_Send` provide no information regarding which **float**s should be printed and which should be used in a computation. So process 1 can use (say) a tag of 0 for the messages that should be printed and a tag of 1 for the messages that should be used in a computation.

The final argument to `MPI_Send` is a communicator. All MPI functions that involve communication have a communicator argument. One of the most important purposes of communicators is to specify communication universes; recall that a communicator is a collection of processes that can send messages to each other. Conversely, a message sent by a process using one communicator cannot be received by a process that's using a different communicator. Since MPI provides functions for creating new communicators, this feature can be used in complex programs to ensure that messages aren't "accidentally received" in the wrong place.

An example will clarify this. Suppose we're studying global climate change, and we've been lucky enough to find two libraries of functions, one for modeling the earth's atmosphere and one for modeling the earth's oceans. Of course, both libraries use MPI. These models were built independently, so they don't communicate with each other, but they do communicate internally. It's our job to write the interface code. One problem we need to solve is ensuring that the messages sent by one library won't be accidentally received by the other. We might be able to work out some scheme with tags: the atmosphere library gets tags $0, 1, \ldots, n-1$, and the ocean library gets tags $n, n+1, \ldots, n+m$. Then each library can use the given range to figure out which tag it should use for which message. However, a much simpler solution is provided by communicators: we simply pass one communicator to the atmosphere library functions and a different communicator to the ocean library functions.

3.1.8 MPI_Recv

The first six arguments to MPI_Recv correspond to the first six arguments of MPI_Send:

```
int MPI_Recv(
      void*          msg_buf_p     /* out */,
      int            buf_size      /* in  */,
      MPI_Datatype   buf_type      /* in  */,
      int            source        /* in  */,
      int            tag           /* in  */,
      MPI_Comm       communicator  /* in  */,
      MPI_Status*    status_p      /* out */);
```

Thus the first three arguments specify the memory available for receiving the message: msg_buf_p points to the block of memory, buf_size determines the number of objects that can be stored in the block, and buf_type indicates the type of the objects. The next three arguments identify the message. The source argument specifies the process from which the message should be received. The tag argument should match the tag argument of the message being sent, and the communicator argument must match the communicator used by the sending process. We'll talk about the status_p argument shortly. In many cases, it won't be used by the calling function, and, as in our "greetings" program, the special MPI constant MPI_STATUS_IGNORE can be passed.

3.1.9 Message matching

Suppose process q calls MPI_Send with

```
MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag,
         send_comm);
```

Also suppose that process r calls MPI_Recv with

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```

Then the message sent by q with the above call to MPI_Send can be received by r with the call to MPI_Recv if

- recv_comm = send_comm,
- recv_tag = send_tag,
- dest = r, and
- src = q.

These conditions aren't quite enough for the message to be *successfully* received, however. The parameters specified by the first three pairs of arguments, send_buf_p/recv_buf_p, send_buf_sz/recv_buf_sz, and send_type/recv_type, must specify compatible buffers. For detailed rules, see the MPI-3 specification [40]. Most of the time, the following rule will suffice:

- If recv_type = send_type and recv_buf_sz $\geq$ send_buf_sz, then the message sent by q can be successfully received by r.

Of course, it can happen that one process is receiving messages from multiple processes, *and* the receiving process doesn't know the order in which the other processes will send the messages. For example, suppose process 0 is doling out work to processes $1, 2, \ldots,$ `comm_sz` $- 1$, and processes $1, 2, \ldots,$ `comm_sz` $- 1$, send their results back to process 0 when they finish the work. If the work assigned to each process takes an unpredictable amount of time, then 0 has no way of knowing the order in which the processes will finish. If process 0 simply receives the results in process rank order—first the results from process 1, then the results from process 2, and so on—and if (say) process `comm_sz`-1 finishes first, it *could* happen that process `comm_sz`-1 could sit and wait for the other processes to finish. To avoid this problem MPI provides a special constant `MPI_ANY_SOURCE` that can be passed to `MPI_Recv`. Then, if process 0 executes the following code, it can receive the results in the order in which the processes finish:

```
for (i = 1; i < comm_sz; i++) {
   MPI_Recv(result, result_sz, result_type, MPI_ANY_SOURCE,
         result_tag, comm, MPI_STATUS_IGNORE);
   Process_result(result);
}
```

Similarly, it's possible that one process can be receiving multiple messages with different tags from another process, and the receiving process doesn't know the order in which the messages will be sent. For this circumstance, MPI provides the special constant `MPI_ANY_TAG` that can be passed to the `tag` argument of `MPI_Recv`.

A couple of points should be stressed in connection with these "wildcard" arguments:

1. Only a receiver can use a wildcard argument. Senders must specify a process rank and a nonnegative tag. So MPI uses a "push" communication mechanism rather than a "pull" mechanism.
2. There is no wildcard for communicator arguments; both senders and receivers must always specify communicators.

3.1.10 The `status_p` argument

If you think about these rules for a minute, you'll notice that a receiver can receive a message without knowing

1. the amount of data in the message,
2. the sender of the message, or
3. the tag of the message.

So how can the receiver find out these values? Recall that the last argument to `MPI_Recv` has type `MPI_Status*`. The MPI type `MPI_Status` is a struct with at least the three members `MPI_SOURCE`, `MPI_TAG`, and `MPI_ERROR`. Suppose our program contains

the definition

```
MPI_Status status;
```

Then after a call to `MPI_Recv`, in which `&status` is passed as the last argument, we can determine the sender and tags by examining the two members:

```
status.MPI_SOURCE
status.MPI_TAG
```

The amount of data that's been received isn't stored in a field that's directly accessible to the application program. However, it can be retrieved with a call to `MPI_Get_count`. For example, suppose that in our call to `MPI_Recv`, the type of the receive buffer is `recv_type` and, once again, we passed in `&status`. Then the call

```
MPI_Get_count(&status, recv_type, &count)
```

will return the number of elements received in the `count` argument. In general, the syntax of `MPI_Get_count` is

```
int MPI_Get_count(
      MPI_Status*  status_p  /* in  */,
      MPI_Datatype type      /* in  */,
      int*         count_p   /* out */);
```

Note that the count isn't directly accessible as a member of the `MPI_Status` variable, simply because it depends on the type of the received data, and, consequently, determining it would probably require a calculation (e.g., (number of bytes received)/(bytes per object)). And if this information isn't needed, we shouldn't waste a calculation determining it.

3.1.11 Semantics of `MPI_Send` and `MPI_Recv`

What exactly happens when we send a message from one process to another? Many of the details depend on the particular system, but we can make a few generalizations. The sending process will assemble the message. For example, it will add the "envelope" information to the actual data being transmitted—the destination process rank, the sending process rank, the tag, the communicator, and some information on the size of the message. Once the message has been assembled, recall from Chapter 2 that there are essentially two possibilities: the sending process can **buffer** the message or it can **block**. If it buffers the message, the MPI system will place the message (data and envelope) into its own internal storage, and the call to `MPI_Send` will return.

Alternatively, if the system blocks, it will wait until it can begin transmitting the message, and the call to `MPI_Send` may not return immediately. Thus if we use `MPI_Send`, when the function returns, we don't actually know whether the message has been transmitted. We only know that the storage we used for the message, the send buffer, is available for reuse by our program. If we need to know that the message has been transmitted, or if we need for our call to `MPI_Send` to return immediately—regardless of whether the message has been sent—MPI provides alternative functions for sending. We'll learn about one of these alternative functions later.

The exact behavior of `MPI_Send` is determined by the MPI implementation. However, typical implementations have a default "cutoff" message size. If the size of a message is less than the cutoff, it will be buffered. If the size of the message is greater than the cutoff, `MPI_Send` will block.

Unlike `MPI_Send`, `MPI_Recv` always blocks until a matching message has been received. So when a call to `MPI_Recv` returns, we know that there is a message stored in the receive buffer (unless there's been an error). There is an alternate method for receiving a message, in which the system checks whether a matching message is available and returns, regardless of whether there is one.

MPI requires that messages be *non-overtaking*. This means that if process q sends two messages to process r, then the first message sent by q must be available to r before the second message. However, there is no restriction on the arrival of messages sent from different processes. That is, if q and t both send messages to r, then even if q sends its message before t sends its message, there is no requirement that q's message become available to r before t's message. This is essentially because MPI can't impose performance on a network. For example, if q happens to be running on a machine on Mars, while r and t are both running on the same machine in San Francisco, and if q sends its message a nanosecond before t sends its message, it would be extremely unreasonable to require that q's message arrive before t's.

3.1.12 Some potential pitfalls

Note that the semantics of `MPI_Recv` suggests a potential pitfall in MPI programming: If a process tries to receive a message and there's no matching send, then the process will block forever. That is, the process will **hang**. So when we design our programs, we need to be sure that every receive has a matching send. Perhaps even more important, we need to be very careful when we're coding that there are no inadvertent mistakes in our calls to `MPI_Send` and `MPI_Recv`. For example, if the tags don't match, or if the rank of the destination process is the same as the rank of the source process, the receive won't match the send, and either a process will hang, or, perhaps worse, the receive may match *another* send.

Similarly, if a call to `MPI_Send` blocks and there's no matching receive, then the sending process can hang. If, on the other hand, a call to `MPI_Send` is buffered and there's no matching receive, then the message will be lost.

3.2 The trapezoidal rule in MPI

Printing messages from processes is all well and good, but we're probably not taking the trouble to learn to write MPI programs just to print messages. So let's take a look at a somewhat more useful program—let's write a program that implements the trapezoidal rule for numerical integration.

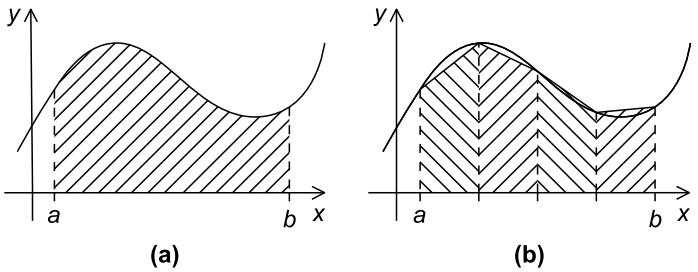

FIGURE 3.3

The trapezoidal rule: (a) area to be estimated and (b) approximate area using trapezoids.

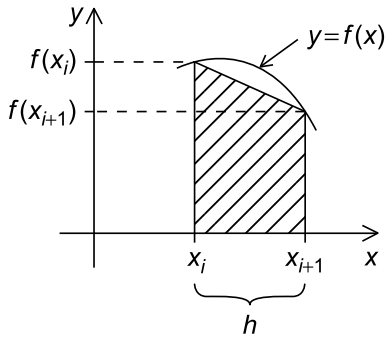

FIGURE 3.4

One trapezoid.

3.2.1 The trapezoidal rule

Recall that we can use the trapezoidal rule to approximate the area between the graph of a function, $y = f(x)$, two vertical lines, and the x-axis. (See Fig. 3.3.) The basic idea is to divide the interval on the x-axis into n equal subintervals. Then we approximate the area lying between the graph and each subinterval by a trapezoid, whose base is the subinterval, whose vertical sides are the vertical lines through the endpoints of the subinterval, and whose fourth side is the secant line joining the points where the vertical lines cross the graph. (See Fig. 3.4.) If the endpoints of the subinterval are x_i and x_{i+1}, then the length of the subinterval is $h = x_{i+1} - x_i$. Also, if the lengths of the two vertical segments are $f(x_i)$ and $f(x_{i+1})$, then the area of the trapezoid is

$$\text{Area of one trapezoid} = \frac{h}{2}[f(x_i) + f(x_{i+1})].$$

Since we chose the n subintervals so that they would all have the same length, we also know that if the vertical lines bounding the region are $x = a$ and $x = b$, then

$$h = \frac{b-a}{n}.$$

Thus if we call the leftmost endpoint x_0, and the rightmost endpoint x_n, we have that

$$x_0 = a,\ x_1 = a + h,\ x_2 = a + 2h,\ \ldots,\ x_{n-1} = a + (n-1)h,\ x_n = b,$$

and the sum of the areas of the trapezoids—our approximation to the total area—is

$$\text{Sum of trapezoid areas} = h[f(x_0)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + f(x_n)/2].$$

Thus, pseudocode for a serial program might look something like this:

```
/* Input:  a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++) {
   x_i = a + i*h;
   approx += f(x_i);
}
approx = h*approx;
```

3.2.2 Parallelizing the trapezoidal rule

It is not the most attractive word, but, as we noted in Chapter 1, people who write parallel programs do use the verb "parallelize" to describe the process of converting a serial program or algorithm into a parallel program.

Recall that we can design a parallel program by using four basic steps:

1. Partition the problem solution into tasks.
2. Identify communication channels between the tasks.
3. Aggregate tasks into composite tasks.
4. Map composite tasks to cores.

In the partitioning phase, we usually try to identify as many tasks as possible. For the trapezoidal rule, we might identify two types of tasks: one type is finding the area of a single trapezoid, and the other is computing the sum of these areas. Then the communication channels will join each of the tasks of the first type to the single task of the second type. (See Fig. 3.5.)

So how can we aggregate the tasks and map them to the cores? Our intuition tells us that the more trapezoids we use, the more accurate our estimate will be. That is, we should use many trapezoids, and we will use many more trapezoids than cores. Thus we need to aggregate the computation of the areas of the trapezoids into groups. A natural way to do this is to split the interval $[a, b]$ up into `comm_sz` subintervals. If `comm_sz` evenly divides n, the number of trapezoids, we can simply apply the trapezoidal rule with $n/$`comm_sz` trapezoids to each of the `comm_sz` subintervals. To finish, we can have one of the processes, say process 0, add the estimates.

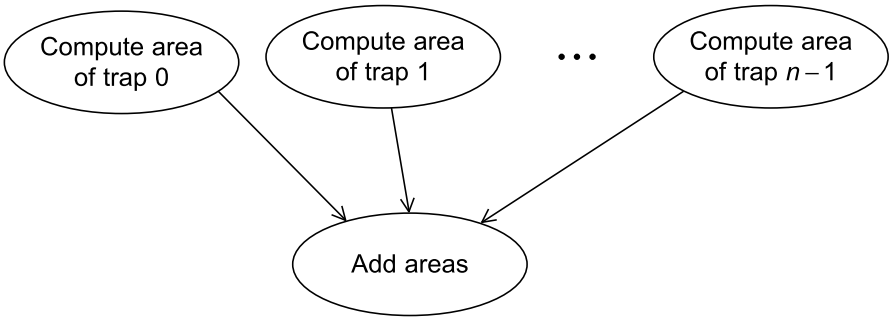

FIGURE 3.5

Tasks and communications for the trapezoidal rule.

Let's make the simplifying assumption that `comm_sz` evenly divides n. Then pseudocode for the program might look something like the following:

```
1   Get a, b, n;
2   h = (b-a)/n;
3   local_n = n/comm_sz;
4   local_a = a + my_rank*local_n*h;
5   local_b = local_a + local_n*h;
6   local_integral = Trap(local_a, local_b, local_n, h);
7   if (my_rank != 0)
8      Send local_integral to process 0;
9   else /* my_rank == 0 */
10     total_integral = local_integral;
11     for (proc = 1; proc < comm_sz; proc++) {
12        Receive local_integral from proc;
13        total_integral += local_integral;
14     }
15  }
16  if (my_rank == 0)
17     print result;
```

Let's defer, for the moment, the issue of input and just "hardwire" the values for a, b, and n. When we do this, we get the MPI program shown in Program 3.2. The `Trap` function is just an implementation of the serial trapezoidal rule. See Program 3.3.

Notice that in our choice of identifiers, we try to differentiate between *local* and *global* variables. **Local** variables are variables whose contents are significant only on the process that's using them. Some examples from the trapezoidal rule program are `local_a`, `local_b`, and `local_n`. Variables whose contents are significant to all the processes are sometimes called **global** variables. Some examples from the trapezoidal rule are `a`, `b`, and `n`. Note that this usage is different from the usage you learned in your introductory programming class, where local variables are private to a single function and global variables are accessible to all the functions. However, no confusion should arise, since the context should make the meaning clear.

```
1  int main(void) {
2     int my_rank, comm_sz, n = 1024, local_n;
3     double a = 0.0, b = 3.0, h, local_a, local_b;
4     double local_int, total_int;
5     int source;
6
7     MPI_Init(NULL, NULL);
8     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
9     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
10
11    h = (b-a)/n;           /* h is the same for all processes */
12    local_n = n/comm_sz;   /* So is the number of trapezoids  */
13
14    local_a = a + my_rank*local_n*h;
15    local_b = local_a + local_n*h;
16    local_int = Trap(local_a, local_b, local_n, h);
17
18    if (my_rank != 0) {
19       MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,
20             MPI_COMM_WORLD);
21    } else {
22       total_int = local_int;
23       for (source = 1; source < comm_sz; source++) {
24          MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
25                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26          total_int += local_int;
27       }
28    }
29
30    if (my_rank == 0) {
31       printf("With n = %d trapezoids, our estimate\n", n);
32       printf("of the integral from %f to %f = %.15e\n",
33           a, b, total_int);
34    }
35    MPI_Finalize();
36    return 0;
37 } /*  main  */
```

Program 3.2: First version of MPI trapezoidal rule.

3.3 Dealing with I/O

Of course, the current version of the parallel trapezoidal rule has a serious deficiency: it will only compute the integral over the interval [0, 3] using 1024 trapezoids. We can edit the code and recompile, but this is quite a bit of work compared to simply typing in three new numbers. We need to address the problem of getting input from

```
1  double Trap(
2        double left_endpt  /* in */,
3        double right_endpt /* in */,
4        int    trap_count  /* in */,
5        double base_len    /* in */) {
6     double estimate, x;
7     int i;
8
9     estimate = (f(left_endpt) + f(right_endpt))/2.0;
10    for (i = 1; i <= trap_count-1; i++) {
11       x = left_endpt + i*base_len;
12       estimate += f(x);
13    }
14    estimate = estimate*base_len;
15
16    return estimate;
17 } /*  Trap  */
```

Program 3.3: `Trap` function in MPI trapezoidal rule.

the user. While we're talking about input to parallel programs, it might be a good idea to also take a look at output. We discussed these two issues in Chapter 2. So if you remember the discussion of nondeterminism and output, you can skip ahead to Section 3.3.2.

3.3.1 Output

In both the "greetings" program and the trapezoidal rule program, we've assumed that process 0 can write to `stdout`, i.e., its calls to `printf` behave as we might expect. Although the MPI standard doesn't specify which processes have access to which I/O devices, virtually all MPI implementations allow *all* the processes in `MPI_COMM_WORLD` full access to `stdout` and `stderr`. So most MPI implementations allow all processes to execute `printf` and `fprintf(stderr, ...)` .

However, most MPI implementations don't provide any automatic scheduling of access to these devices. That is, if multiple processes are attempting to write to, say, `stdout`, the order in which the processes' output appears will be unpredictable. Indeed, it can even happen that the output of one process will be interrupted by the output of another process.

For example, suppose we try to run an MPI program in which each process simply prints a message. (See Program 3.4.) On our cluster, if we run the program with five processes, it often produces the "expected" output:

```
Proc 0 of 5 > Does anyone have a toothpick?
Proc 1 of 5 > Does anyone have a toothpick?
Proc 2 of 5 > Does anyone have a toothpick?
```

```
#include <stdio.h>
#include <mpi.h>

int main(void) {
   int my_rank, comm_sz;

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   printf("Proc %d of %d > Does anyone have a toothpick?\n",
          my_rank, comm_sz);

   MPI_Finalize();
   return 0;
}  /* main */
```

Program 3.4: Each process just prints a message.

```
Proc 3 of 5 > Does anyone have a toothpick?
Proc 4 of 5 > Does anyone have a toothpick?
```

However, when we run it with six processes, the order of the output lines is unpredictable:

```
Proc 0 of 6 > Does anyone have a toothpick?
Proc 1 of 6 > Does anyone have a toothpick?
Proc 2 of 6 > Does anyone have a toothpick?
Proc 5 of 6 > Does anyone have a toothpick?
Proc 3 of 6 > Does anyone have a toothpick?
Proc 4 of 6 > Does anyone have a toothpick?
```

or

```
Proc 0 of 6 > Does anyone have a toothpick?
Proc 1 of 6 > Does anyone have a toothpick?
Proc 2 of 6 > Does anyone have a toothpick?
Proc 4 of 6 > Does anyone have a toothpick?
Proc 3 of 6 > Does anyone have a toothpick?
Proc 5 of 6 > Does anyone have a toothpick?
```

The reason this happens is that the MPI processes are "competing" for access to the shared output device, `stdout`, and it's impossible to predict the order in which the processes' output will be queued up. Such a competition results in **nondeterminism**. That is, the actual output will vary from one run to the next.

In any case, if we don't want output from different processes to appear in a random order, it's up to us to modify our program accordingly. For example, we can have each

process other than 0 send its output to process 0, and process 0 can print the output in process rank order. This is exactly what we did in the "greetings" program.

3.3.2 Input

Unlike output, most MPI implementations only allow process 0 in `MPI_COMM_WORLD` access to `stdin`. This makes sense: If multiple processes have access to `stdin`, which process should get which parts of the input data? Should process 0 get the first line? process 1 get the second? Or should process 0 get the first character?

So, to write MPI programs that can use `scanf`, we need to branch on process rank, with process 0 reading in the data, and then sending it to the other processes. For example, we might write the `Get_input` function shown in Program 3.5 for our parallel trapezoidal rule program. In this function, process 0 simply reads in the values

```
1  void Get_input(
2        int       my_rank   /* in  */,
3        int       comm_sz   /* in  */,
4        double*   a_p       /* out */,
5        double*   b_p       /* out */,
6        int*      n_p       /* out */) {
7     int dest;
8
9     if (my_rank == 0) {
10       printf("Enter a, b, and n\n");
11       scanf("%lf %lf %d", a_p, b_p, n_p);
12       for (dest = 1; dest < comm_sz; dest++) {
13          MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
14          MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
15          MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
16       }
17    } else { /* my_rank != 0 */
18       MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
19             MPI_STATUS_IGNORE);
20       MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
21             MPI_STATUS_IGNORE);
22       MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
23             MPI_STATUS_IGNORE);
24    }
25 }  /* Get_input */
```

Program 3.5: A function for reading user input.

for a, b, and n, and sends all three values to each process. So this function uses the same basic communication structure as the "greetings" program, except that now process 0 is sending to each process, while the other processes are receiving.

To use this function, we can simply insert a call to it inside our main function, being careful to put it after we've initialized `my_rank` and `comm_sz`:

```
. . .
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

Get_input(my_rank, comm_sz, &a, &b, &n);

h = (b-a)/n;
. . .
```

3.4 Collective communication

If we pause for a moment and think about our trapezoidal rule program, we can find several things that we might be able to improve on. One of the most obvious is the "global sum" after each process has computed its part of the integral. If we hire eight workers to, say, build a house, we might feel that we weren't getting our money's worth if seven of the workers told the first what to do, and then the seven collected their pay and went home. But this is very similar to what we're doing in our global sum. Each process with rank greater than 0 is "telling process 0 what to do" and then quitting. That is, each process with rank greater than 0 is, in effect, saying "add this number into the total." Process 0 is doing nearly all the work in computing the global sum, while the other processes are doing almost nothing. Sometimes it does happen that this is the best we can do in a parallel program, but if we imagine that we have eight students, each of whom has a number and we want to find the sum of all eight numbers, we can certainly come up with a more equitable distribution of the work than having seven of the eight give their numbers to one of the students and having the first do the addition.

3.4.1 Tree-structured communication

As we already saw in Chapter 1, we might use a "binary tree structure," like that illustrated in Fig. 3.6. In this diagram, initially students or processes 1, 3, 5, and 7 send their values to processes 0, 2, 4, and 6, respectively. Then processes 0, 2, 4, and 6 add the received values to their original values, and the process is repeated twice:

1. **a.** Processes 2 and 6 send their new values to processes 0 and 4, respectively.
   b. Processes 0 and 4 add the received values into their new values.
2. **a.** Process 4 sends its newest value to process 0.
   b. Process 0 adds the received value to its newest value.

This solution may not seem ideal, since half the processes (1, 3, 5, and 7) are doing the same amount of work that they did in the original scheme. However, if

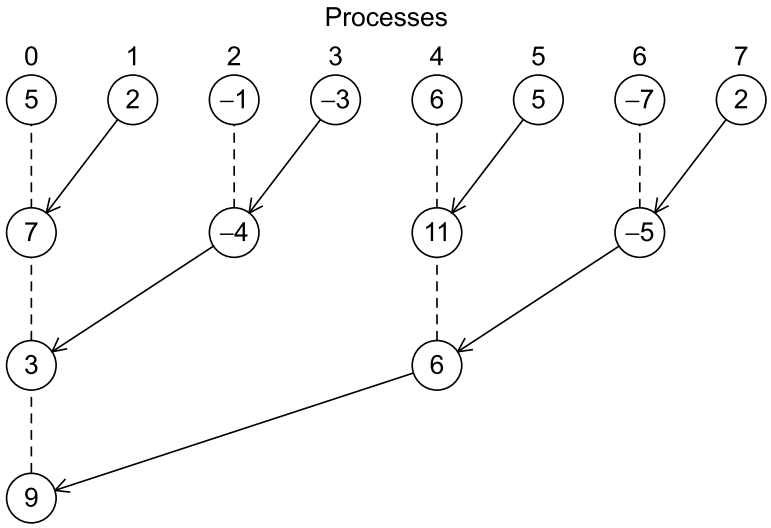

FIGURE 3.6

A tree-structured global sum.

you think about it, the original scheme required `comm_sz` $- 1 =$ seven receives and seven adds by process 0, while the new scheme only requires three, and all the other processes do no more than two receives and adds. Furthermore, the new scheme has the property that a lot of the work is done concurrently by different processes. For example, in the first phase, the receives and adds by processes 0, 2, 4, and 6 can all take place simultaneously. So, if the processes start at roughly the same time, the total time required to compute the global sum will be the time required by process 0, i.e., three receives and three additions. So we've reduced the overall time by more than 50%. Furthermore, if we use more processes, we can do even better. For example, if `comm_sz` $= 1024$, then the original scheme requires process 0 to do 1023 receives and additions, while it can be shown (Exercise 3.5) that the new scheme requires process 0 to do only 10 receives and additions. This improves the original scheme by more than a factor of 100!

You may be thinking to yourself, this is all well and good, but coding this tree-structured global sum looks like it would take a quite a bit of work, and you'd be right. (See Programming Assignment 3.3.) In fact, the problem may be even harder. For example, it's perfectly feasible to construct a tree-structured global sum that uses different "process-pairings." For example, we might pair 0 and 4, 1 and 5, 2 and 6, and 3 and 7 in the first phase. Then we could pair 0 and 2, and 1 and 3 in the second, and 0 and 1 in the final. (See Fig. 3.7.) Of course, there are many other possibilities. How can we decide which is the best? Do we need to code each alternative and evaluate their performance? If we do, is it possible that one method works best for "small"

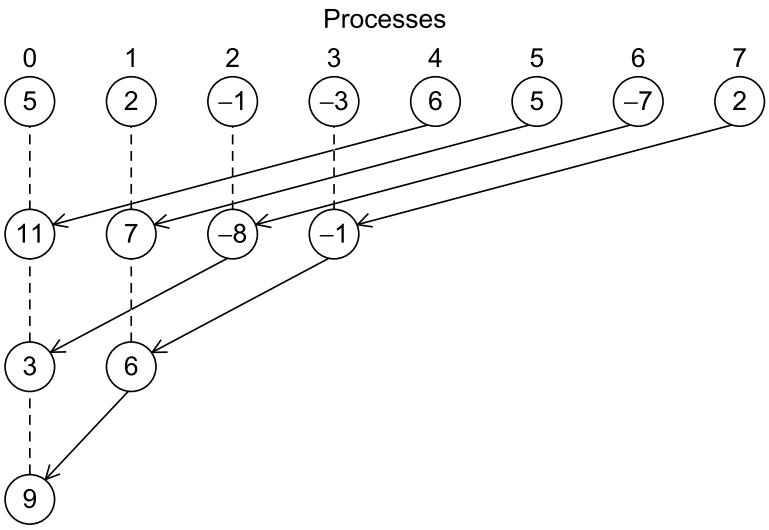

FIGURE 3.7

An alternative tree-structured global sum.

trees, while another works best for "large" trees? Even worse, one approach might work best on system A, while another might work best on system B.

3.4.2 MPI_Reduce

With virtually limitless possibilities, it's unreasonable to expect each MPI programmer to write an optimal global-sum function, so MPI specifically protects programmers against this trap of endless optimization by requiring that MPI implementations include implementations of global sums. This places the burden of optimization on the developer of the MPI implementation, rather than the application developer. The assumption here is that the developer of the MPI implementation should know enough about both the hardware and the system software so that she can make better decisions about implementation details.

Now a "global-sum function" will obviously require communication. However, unlike the `MPI_Send`-`MPI_Recv` pair, the global-sum function may involve more than two processes. In fact, in our trapezoidal rule program it will involve all the processes in `MPI_COMM_WORLD`. In MPI parlance, communication functions that involve all the processes in a communicator are called **collective communications**. To distinguish between collective communications and functions, such as `MPI_Send` and `MPI_Recv`, `MPI_Send` and `MPI_Recv` are often called **point-to-point** communications.

In fact, global-sum is just a special case of an entire class of collective communications. For example, it might happen that instead of finding the sum of a collection of `comm_sz` numbers distributed among the processes, we want to find the maximum or the minimum or the product, or any one of many other possibilities. So MPI general-

Table 3.2 Predefined Reduction Operators in MPI.

Operation Value	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

ized the global-sum function so that any one of these possibilities can be implemented with a single function:

```
int MPI_Reduce(
      void*         input_data_p   /* in  */,
      void*         output_data_p  /* out */,
      int           count          /* in  */,
      MPI_Datatype  datatype       /* in  */,
      MPI_Op        operator       /* in  */,
      int           dest_process   /* in  */,
      MPI_Comm      comm           /* in  */);
```

The key to the generalization is the fifth argument, `operator`. It has type `MPI_Op`, which is a predefined MPI type—like `MPI_Datatype` and `MPI_Comm`. There are a number of predefined values in this type—see Table 3.2. It's also possible to define your own operators—for details, see the MPI-3 Standard [40].

The operator we want is `MPI_SUM`. Using this value for the `operator` argument, we can replace the code in Lines 18–28 of Program 3.2 with the single function call

```
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM,
      0, MPI_COMM_WORLD);
```

One point worth noting is that by using a `count` argument greater than 1, `MPI_Reduce` can operate on arrays instead of scalars. So the following code could be used to add a collection of N-dimensional vectors, one per process:

```
double local_x[N], sum[N];
. . .
MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
```

Table 3.3 Multiple Calls to `MPI_Reduce`.

Time	Process 0	Process 1	Process 2
0	`a = 1; c = 2`	`a = 1; c = 2`	`a = 1; c = 2`
1	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`
2	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`

3.4.3 Collective vs. point-to-point communications

It's important to remember that collective communications differ in several ways from point-to-point communications:

1. *All* the processes in the communicator must call the same collective function. For example, a program that attempts to match a call to `MPI_Reduce` on one process with a call to `MPI_Recv` on another process is erroneous, and, in all likelihood, the program will hang or crash.
2. The arguments passed by each process to an MPI collective communication must be "compatible." For example, if one process passes in 0 as the `dest_process` and another passes in 1, then the outcome of a call to `MPI_Reduce` is erroneous, and, once again, the program is likely to hang or crash.
3. The `output_data_p` argument is only used on `dest_process`. However, all of the processes still need to pass in an actual argument corresponding to `output_data_p`, even if it's just `NULL`.
4. Point-to-point communications are matched on the basis of tags and communicators. Collective communications don't use tags. So they're matched solely on the basis of the communicator and the *order* in which they're called. As an example, consider the calls to `MPI_Reduce` shown in Table 3.3. Suppose that each process calls `MPI_Reduce` with operator `MPI_SUM`, and destination process 0. At first glance, it might seem that after the two calls to `MPI_Reduce`, the value of `b` will be 3, and the value of `d` will be 6. However, the names of the memory locations are irrelevant to the matching of the calls to `MPI_Reduce`. The *order* of the calls will determine the matching, so the value stored in `b` will be $1 + 2 + 1 = 4$, and the value stored in `d` will be $2 + 1 + 2 = 5$.

A final caveat: it might be tempting to call `MPI_Reduce` using the same buffer for both input and output. For example, if we wanted to form the global sum of `x` on each process and store the result in `x` on process 0, we might try calling

```
MPI_Reduce(&x, &x, 1, MPI_DOUBLE, MPI_SUM, 0, comm);
```

However, this call is illegal in MPI. So its result will be unpredictable: it might produce an incorrect result, it might cause the program to crash; it might even produce a correct result. It's illegal, because it involves **aliasing** of an output argument. Two

arguments are aliased if they refer to the same block of memory, and MPI prohibits aliasing of arguments if one of them is an output or input/output argument. This is because the MPI Forum wanted to make the Fortran and C versions of MPI as similar as possible, and Fortran prohibits aliasing. In some instances MPI provides an alternative construction that effectively avoids this restriction. See Subsection 7.1.9 for an example.

3.4.4 MPI_Allreduce

In our trapezoidal rule program, we just print the result. So it's perfectly natural for only one process to get the result of the global sum. However, it's not difficult to imagine a situation in which *all* of the processes need the result of a global sum to complete some larger computation. In this situation, we encounter some of the same problems we encountered with our original global sum. For example, if we use a tree to compute a global sum, we might "reverse" the branches to distribute the global sum (see Fig. 3.8). Alternatively, we might have the processes *exchange* partial results instead of using one-way communications. Such a communication pattern is sometimes called a *butterfly*. (See Fig. 3.9.) Once again, we don't want to have to decide on which structure to use, or how to code it for optimal performance. Fortunately, MPI provides a variant of `MPI_Reduce` that will store the result on all the processes in the communicator:

```
int MPI_Allreduce(
          void*        input_data_p    /* in  */,
          void*        output_data_p   /* out */,
          int          count           /* in  */,
          MPI_Datatype datatype        /* in  */,
          MPI_Op       operator        /* in  */,
          MPI_Comm     comm            /* in  */);
```

The argument list is identical to that for `MPI_Reduce`, except that there is no `dest_process` since all the processes should get the result.

3.4.5 Broadcast

If we can improve the performance of the global sum in our trapezoidal rule program by replacing a loop of receives on process 0 with a tree structured communication, we ought to be able to do something similar with the distribution of the input data. In fact, if we simply "reverse" the communications in the tree-structured global sum in Fig. 3.6, we obtain the tree-structured communication shown in Fig. 3.10, and we can use this structure to distribute the input data. A collective communication in which data belonging to a single process is sent to all of the processes in the communicator is called a **broadcast**, and you've probably guessed that MPI provides a broadcast function:

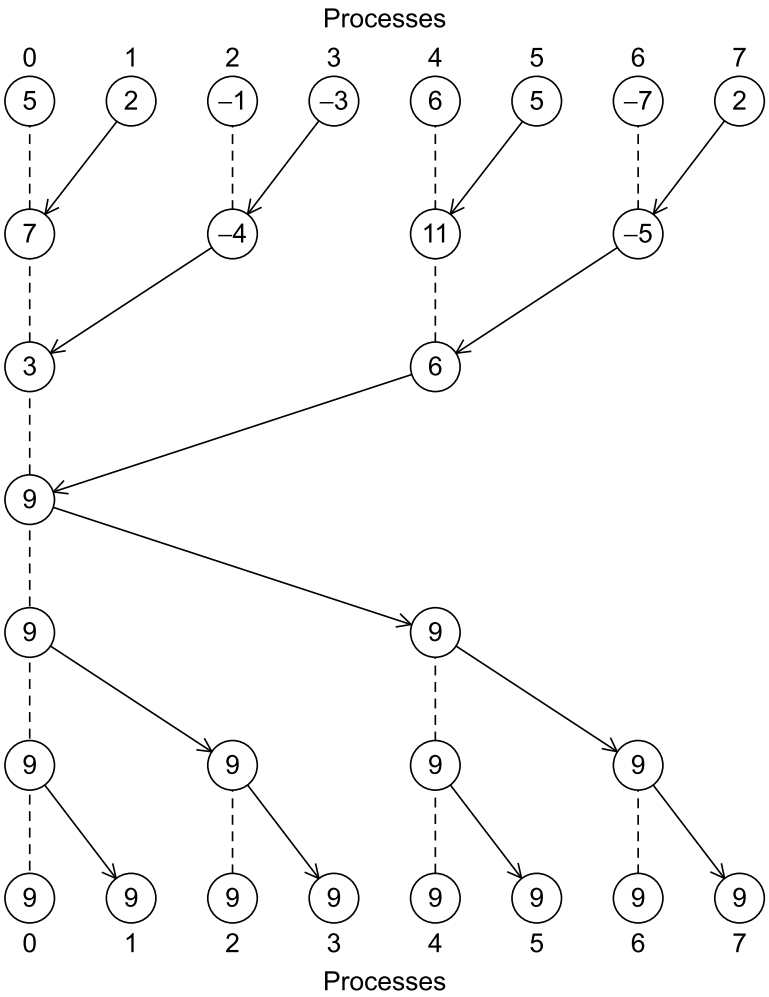

FIGURE 3.8

A global sum followed by distribution of the result.

```
int MPI_Bcast(
      void*         data_p        /* in/out */,
      int           count         /* in     */,
      MPI_Datatype  datatype      /* in     */,
      int           source_proc   /* in     */,
      MPI_Comm      comm          /* in     */);
```

The process with rank `source_proc` sends the contents of the memory referenced by `data_p` to all the processes in the communicator `comm`.

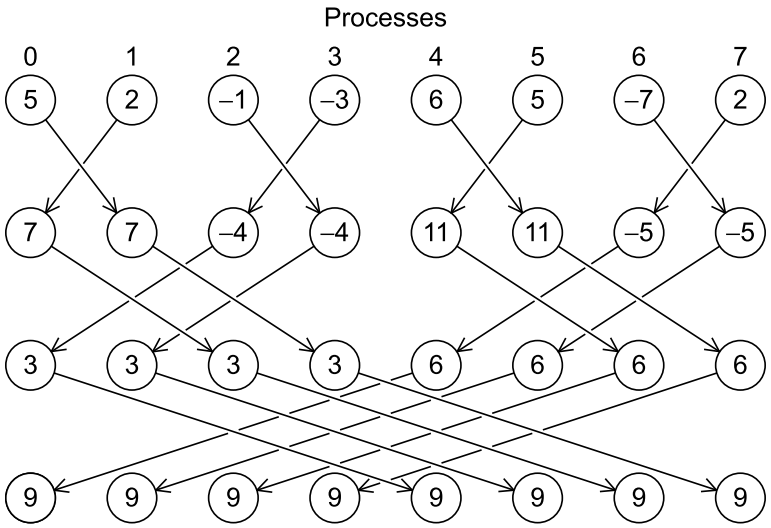

FIGURE 3.9

A butterfly-structured global sum.

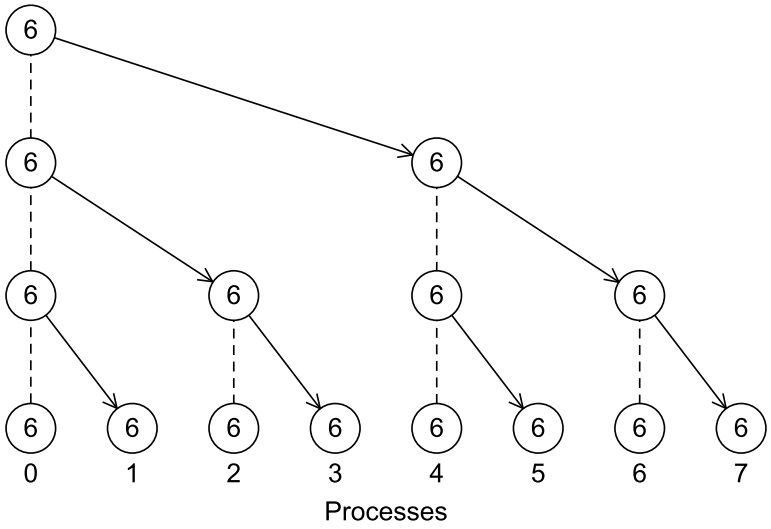

FIGURE 3.10

A tree-structured broadcast.

Program 3.6 shows how to modify the `Get_input` function shown in Program 3.5 so that it uses `MPI_Bcast`, instead of `MPI_Send` and `MPI_Recv`.

```
1  void Get_input(
2        int        my_rank  /* in  */,
3        int        comm_sz  /* in  */,
4        double*    a_p      /* out */,
5        double*    b_p      /* out */,
6        int*       n_p      /* out */) {
7
8     if (my_rank == 0) {
9        printf("Enter a, b, and n\n");
10       scanf("%lf %lf %d", a_p, b_p, n_p);
11    }
12    MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
13    MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
14    MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
15 }  /* Get_input */
```

Program 3.6: A version of `Get_input` that uses `MPI_Bcast`.

Recall that in serial programs an in/out argument is one whose value is both used and changed by the function. For `MPI_Bcast`, however, the `data_p` argument is an input argument on the process with rank `source_proc` and an output argument on the other processes. Thus when an argument to a collective communication is labeled in/out, it's possible that it's an input argument on some processes and an output argument on other processes.

3.4.6 Data distributions

Suppose we want to write a function that computes a vector sum:

$$\begin{aligned}\mathbf{x} + \mathbf{y} &= (x_0, x_1, \ldots, x_{n-1}) + (y_0, y_1, \ldots, y_{n-1}) \\ &= (x_0 + y_0, x_1 + y_1, \ldots, x_{n-1} + y_{n-1}) \\ &= (z_0, z_1, \ldots, z_{n-1}) \\ &= \mathbf{z}\end{aligned}$$

If we implement the vectors as arrays of, say, **double**s, we could implement serial vector addition with the code shown in Program 3.7.

How could we implement this using MPI? The work consists of adding the individual components of the vectors, so we might specify that the tasks are just the additions of corresponding components. Then there is no communication between the tasks, and the problem of parallelizing vector addition boils down to aggregating the tasks and assigning them to the cores. If the number of components is n and we have `comm_sz` cores or processes, let's assume that n is evenly divisible by `comm_sz` and define `local_n` $= n/$`comm_sz`. Then we can simply assign blocks of `local_n` consecutive components to each process. The four columns on the left of Table 3.4 show an

```
1  void Vector_sum(double x[], double y[], double z[], int n) {
2     int i;
3
4     for (i = 0; i < n; i++)
5        z[i] = x[i] + y[i];
6  }  /* Vector_sum */
```

Program 3.7: A serial implementation of vector addition.

Table 3.4 Different partitions of a 12-component vector among 3 processes.

	Components											
	Block				Cyclic				Block-cyclic Blocksize = 2			
Process												
0	0	1	2	3	0	3	6	9	0	1	6	7
1	4	5	6	7	1	4	7	10	2	3	8	9
2	8	9	10	11	2	5	8	11	4	5	10	11

example when $n = 12$ and `comm_sz` $= 3$. This is often called a **block partition** of the vector.

An alternative to a block partition is a **cyclic partition**. In a cyclic partition, we assign the components in a round-robin fashion. The four columns in the middle of Table 3.4 show an example when $n = 12$ and `comm_sz` $= 3$. So process 0 gets component 0, process 1 gets component 1, process 2 gets component 2, process 0 gets component 3, and so on.

A third alternative is a **block-cyclic partition**. The idea here is that instead of using a cyclic distribution of individual components, we use a cyclic distribution of *blocks* of components. So a block-cyclic distribution isn't fully specified until we decide how large the blocks are. If `comm_sz` $= 3$, $n = 12$, and the blocksize $b = 2$, an example is shown in the four columns on the right of Table 3.4.

Once we've decided how to partition the vectors, it's easy to write a parallel vector addition function: each process simply adds its assigned components. Furthermore, regardless of the partition, each process will have `local_n` components of the vector, and, to save on storage, we can just store these on each process as an array of `local_n` elements. Thus each process will execute the function shown in Program 3.8. Although the names of the variables have been changed to emphasize the fact that the function is operating on only the process's portion of the vector, this function is virtually identical to the original serial function.

3.4.7 Scatter

Now suppose we want to test our vector addition function. It would be convenient to be able to read the dimension of the vectors and then read in the vectors **x** and **y**. We already know how to read in the dimension of the vectors: process 0 can prompt the

```
1  void Parallel_vector_sum(
2        double  local_x[]  /* in  */,
3        double  local_y[]  /* in  */,
4        double  local_z[]  /* out */,
5        int     local_n    /* in  */) {
6     int local_i;
7
8     for (local_i = 0; local_i < local_n; local_i++)
9        local_z[local_i] = local_x[local_i] + local_y[local_i];
10 }  /* Parallel_vector_sum */
```

Program 3.8: A parallel implementation of vector addition.

user, read in the value, and broadcast the value to the other processes. We might try something similar with the vectors: process 0 could read them in and broadcast them to the other processes. However, this could be very wasteful. If there are 10 processes and the vectors have 10,000 components, then each process will need to allocate storage for vectors with 10,000 components, when it is only operating on subvectors with 1000 components. If, for example, we use a block distribution, it would be better if process 0 sent only components 1000 to 1999 to process 1, components 2000 to 2999 to process 2, etc. Using this approach processes 1 to 9 would only need to allocate storage for the components they're actually using.

Thus we might try writing a function that reads in an entire vector on process 0, but only sends the needed components to each of the other processes. For the communication MPI provides just such a function:

```
int MPI_Scatter(
      void*         send_buf_p  /* in  */,
      int           send_count  /* in  */,
      MPI_Datatype  send_type   /* in  */,
      void*         recv_buf_p  /* out */,
      int           recv_count  /* in  */,
      MPI_Datatype  recv_type   /* in  */,
      int           src_proc    /* in  */,
      MPI_Comm      comm        /* in  */);
```

If the communicator `comm` contains `comm_sz` processes, then `MPI_Scatter` divides the data referenced by `send_buf_p` into `comm_sz` pieces—the first piece goes to process 0, the second to process 1, the third to process 2, and so on. For example, suppose we're using a block distribution and process 0 has read in all of an n-component vector into `send_buf_p`. Then process 0 will get the first `local_n` $= n/$`comm_sz` components, process 1 will get the next `local_n` components, and so on. Each process should pass its local vector as the `recv_buf_p` argument, and the `recv_count` argument should be `local_n`. Both `send_type` and `recv_type` should be `MPI_DOUBLE`, and `src_proc` should be 0. Perhaps surprisingly, `send_count` should also be `local_n`—`send_count` is the

amount of data going to each process; it's *not* the amount of data in the memory referred to by `send_buf_p`. If we use a block distribution and `MPI_Scatter`, we can read in a vector using the function `Read_vector` shown in Program 3.9.

```
1  void Read_vector(
2        double     local_a[]   /* out */,
3        int        local_n     /* in  */,
4        int        n           /* in  */,
5        char       vec_name[]  /* in  */,
6        int        my_rank     /* in  */,
7        MPI_Comm   comm        /* in  */) {
8
9     double* a = NULL;
10    int i;
11
12    if (my_rank == 0) {
13       a = malloc(n*sizeof(double));
14       printf("Enter the vector %s\n", vec_name);
15       for (i = 0; i < n; i++)
16          scanf("%lf", &a[i]);
17       MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n,
18             MPI_DOUBLE, 0, comm);
19       free(a);
20    } else {
21       MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n,
22             MPI_DOUBLE, 0, comm);
23    }
24 } /* Read_vector */
```

Program 3.9: A function for reading and distributing a vector.

One point to note here is that `MPI_Scatter` sends the first block of `send_count` objects to process 0, the next block of `send_count` objects to process 1, and so on. So this approach to reading and distributing the input vectors will only be suitable if we're using a block distribution and n, the number of components in the vectors, is evenly divisible by `comm_sz`. We'll discuss how we might deal with a cyclic or block cyclic distribution in Section 3.5, and we'll look at dealing with the case in which n is not evenly divisible by `comm_sz` in Exercise 3.13.

3.4.8 Gather

Of course, our test program will be useless unless we can see the result of our vector addition. So we need to write a function for printing out a distributed vector. Our function can collect all of the components of the vector onto process 0, and then process 0 can print all of the components. The communication in this function can be carried out by `MPI_Gather`:

```
int MPI_Gather(
      void*          send_buf_p  /* in  */,
      int            send_count  /* in  */,
      MPI_Datatype   send_type   /* in  */,
      void*          recv_buf_p  /* out */,
      int            recv_count  /* in  */,
      MPI_Datatype   recv_type   /* in  */,
      int            dest_proc   /* in  */,
      MPI_Comm       comm        /* in  */);
```

The data stored in the memory referred to by `send_buf_p` on process 0 is stored in the first block in `recv_buf_p`, the data stored in the memory referred to by `send_buf_p` on process 1 is stored in the second block referred to by `recv_buf_p`, and so on. So, if we're using a block distribution, we can implement our distributed vector print function, as shown in Program 3.10. Note that `recv_count` is the number of data items received from *each* process, not the total number of data items received.

```
1  void Print_vector(
2        double     local_b[]  /* in */,
3        int        local_n    /* in */,
4        int        n          /* in */,
5        char       title[]    /* in */,
6        int        my_rank    /* in */,
7        MPI_Comm   comm       /* in */) {
8
9     double* b = NULL;
10    int i;
11
12    if (my_rank == 0) {
13       b = malloc(n*sizeof(double));
14       MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n,
15             MPI_DOUBLE, 0, comm);
16       printf("%s\n", title);
17       for (i = 0; i < n; i++)
18          printf("%f ", b[i]);
19       printf("\n");
20       free(b);
21    } else {
22       MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n,
23             MPI_DOUBLE, 0, comm);
24    }
25 } /* Print_vector */
```

Program 3.10: A function for printing a distributed vector.

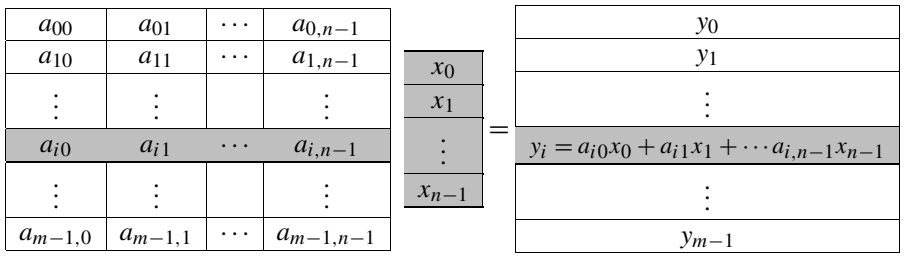

FIGURE 3.11

Matrix-vector multiplication.

The restrictions on the use of `MPI_Gather` are similar to those on the use of `MPI_Scatter`: our print function will only work correctly with vectors using a block distribution in which each block has the same size.

3.4.9 Allgather

As a final example, let's look at how we might write an MPI function that multiplies a matrix by a vector. Recall that if $A = (a_{ij})$ is an $m \times n$ matrix and $\mathbf{x}$ is a vector with n components, then $\mathbf{y} = A\mathbf{x}$ is a vector with m components, and we can find the ith component of y by forming the dot product of the ith row of A with $\mathbf{x}$:

$$y_i = a_{i0}x_0 + a_{i1}x_1 + a_{i2}x_2 + \cdots a_{i,n-1}x_{n-1}.$$

(See Fig. 3.11.)

So we might write pseudocode for serial matrix multiplication as follows:

```
/* For each row of A */
for (i = 0; i < m; i++) {
   /* Form dot product of ith row with x */
   y[i] = 0.0;
   for (j = 0; j < n; j++)
      y[i] += A[i][j]*x[j];
}
```

In fact, this could be actual C code. However, there are some peculiarities in the way that C programs deal with two-dimensional arrays (see Exercise 3.14). So C programmers frequently use one-dimensional arrays to "simulate" two-dimensional arrays. The most common way to do this is to list the rows one after another. For example, the two-dimensional array

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{pmatrix}$$

would be stored as the one-dimensional array

$$0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11.$$

In this example, if we start counting rows and columns from 0, then the element stored in row 2 and column 1—the 9—in the two-dimensional array is located in position $2 \times 4 + 1 = 9$ in the one-dimensional array. More generally, if our array has n columns, when we use this scheme, we see that the element stored in row i and column j is located in position $i \times n + j$ in the one-dimensional array.

Using this one-dimensional scheme, we get the C function shown in Program 3.11.

```
1  void Mat_vect_mult(
2        double A[]  /* in  */,
3        double x[]  /* in  */,
4        double y[]  /* out */,
5        int    m    /* in  */,
6        int    n    /* in  */) {
7     int i, j;
8
9     for (i = 0; i < m; i++) {
10       y[i] = 0.0;
11       for (j = 0; j < n; j++)
12          y[i] += A[i*n+j]*x[j];
13    }
14 }  /* Mat_vect_mult */
```

Program 3.11: Serial matrix-vector multiplication.

Now let's see how we might parallelize this function. An individual task can be the multiplication of an element of A by a component of $\mathbf{x}$ and the addition of this product into a component of $\mathbf{y}$. That is, each execution of the statement

```
y[i] += A[i*n+j]*x[j];
```

is a task, so we see that if `y[i]` is assigned to process q, then it would be convenient to also assign row i of `A` to process q. This suggests that we partition `A` by *rows*. We could partition the rows using a block distribution, a cyclic distribution, or a block-cyclic distribution. In MPI it's easiest to use a block distribution. So let's use a block distribution of the rows of `A`, and, as usual, assume that `comm_sz` evenly divides m, the number of rows.

We are distributing `A` by rows so that the computation of `y[i]` will have all of the needed elements of `A`, so we should distribute `y` by blocks. That is, if the ith row of A is assigned to process q, then the ith component of `y` should also be assigned to process q.

Now the computation of `y[i]` involves all the elements in the *i*th row of `A` and *all* the components of `x`. So we could minimize the amount of communication by simply assigning all of `x` to each process. However, in actual applications—especially when the matrix is square—it's often the case that a program using matrix-vector multiplication will execute the multiplication many times, and the result vector **y** from one multiplication will be the input vector **x** for the next iteration. In practice, then, we usually assume that the distribution for `x` is the same as the distribution for `y`.

So if `x` has a block distribution, how can we arrange that each process has access to *all* the components of *x* before we execute the following loop?

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

Using the collective communications we're already familiar with, we could execute a call to `MPI_Gather`, followed by a call to `MPI_Bcast`. This would, in all likelihood, involve two tree-structured communications, and we may be able to do better by using a butterfly. So, once again, MPI provides a single function:

```
int MPI_Allgather(
      void*          send_buf_p  /* in  */,
      int            send_count  /* in  */,
      MPI_Datatype   send_type   /* in  */,
      void*          recv_buf_p  /* out */,
      int            recv_count  /* in  */,
      MPI_Datatype   recv_type   /* in  */,
      MPI_Comm       comm        /* in  */);
```

This function concatenates the contents of each process's `send_buf_p` and stores this in each process's `recv_buf_p`. As usual, `recv_count` is the amount of data being received from *each* process. So in most cases, `recv_count` will be the same as `send_count`.

We can now implement our parallel matrix-vector multiplication function as shown in Program 3.12. If this function is called many times, we can improve performance by allocating `x` once in the calling function and passing it as an additional argument.

3.5 MPI-derived datatypes

In virtually all distributed-memory systems, communication can be *much* more expensive than local computation. For example, sending a **double** from one node to another will take far longer than adding two **double**s stored in the local memory of a node. Furthermore, the cost of sending a fixed amount of data in multiple messages is usually much greater than the cost of sending a single message with the same amount of data. For example, we would expect the following pair of **for** loops to be much slower than the single send/receive pair:

```
1  void Mat_vect_mult(
2        double     local_A[]  /* in  */,
3        double     local_x[]  /* in  */,
4        double     local_y[]  /* out */,
5        int        local_m    /* in  */,
6        int        n          /* in  */,
7        int        local_n    /* in  */,
8        MPI_Comm   comm       /* in  */) {
9     double* x;
10    int local_i, j;
11    int local_ok = 1;
12
13    x = malloc(n*sizeof(double));
14    MPI_Allgather(local_x, local_n, MPI_DOUBLE,
15          x, local_n, MPI_DOUBLE, comm);
16
17    for (local_i = 0; local_i < local_m; local_i++) {
18       local_y[local_i] = 0.0;
19       for (j = 0; j < n; j++)
20          local_y[local_i] += local_A[local_i*n+j]*x[j];
21    }
22    free(x);
23 }  /* Mat_vect_mult */
```

Program 3.12: An MPI matrix-vector multiplication function.

```
double x[1000];
. . .
if (my_rank == 0)
   for (i = 0; i < 1000; i++)
      MPI_Send(&x[i], 1, MPI_DOUBLE, 1, 0, comm);
else /* my_rank == 1 */
   for (i = 0; i < 1000; i++)
      MPI_Recv(&x[i], 1, MPI_DOUBLE, 0, 0, comm, &status);

if (my_rank == 0)
   MPI_Send(x, 1000, MPI_DOUBLE, 1, 0, comm);
else  /* my_rank == 1 */
   MPI_Recv(x, 1000, MPI_DOUBLE, 0, 0, comm, &status);
```

In fact, on one of our systems, the code with the loops of sends and receives takes nearly 50 times longer. On another system, the code with the loops takes more than 100 times longer. Thus if we can reduce the total number of messages we send, we're likely to improve the performance of our programs.

MPI provides three basic approaches to consolidating data that might otherwise require multiple messages: the `count` argument to the various communication

functions, derived datatypes, and MPI_Pack/Unpack. We've already seen the count argument—it can be used to group contiguous array elements into a single message. In this section, we'll discuss one method for building derived datatypes. In the Exercises, we'll take a look at some other methods for building derived datatypes and MPI_Pack/Unpack.

In MPI, a **derived datatype** can be used to represent any collection of data items in memory by storing both the types of the items and their relative locations in memory. The idea here is that if a function that sends data knows the types and the relative locations in memory of a collection of data items, it can collect the items from memory before they are sent. Similarly, a function that receives data can distribute the items into their correct destinations in memory when they're received. As an example, in our trapezoidal rule program, we needed to call MPI_Bcast three times: once for the left endpoint a, once for the right endpoint b, and once for the number of trapezoids n. As an alternative, we could build a single derived datatype that consists of two **double**s and one **int**. If we do this, we'll only need one call to MPI_Bcast. On process 0, a, b, and n will be sent with the one call, while on the other processes, the values will be received with the call.

Formally, a derived datatype consists of a sequence of basic MPI datatypes together with a *displacement* for each of the datatypes. In our trapezoidal rule example, suppose that on process 0 the variables a, b, and n are stored in memory locations with the following addresses:

Variable	Address
a	24
b	40
n	48

Then the following derived datatype could represent these data items:

$$\{(\texttt{MPI\_DOUBLE}, 0), (\texttt{MPI\_DOUBLE}, 16), (\texttt{MPI\_INT}, 24)\}.$$

The first element of each pair corresponds to the type of the data, and the second element of each pair is the displacement of the data element from the *beginning* of the type. We've assumed that the type begins with a, so it has displacement 0, and the other elements have displacements measured in bytes, from a: b is $40 - 24 = 16$ bytes beyond the start of a, and n is $48 - 24 = 24$ bytes beyond the start of a.

We can use MPI_Type_create_struct to build a derived datatype that consists of individual elements that have different basic types:

```
int MPI_Type_create_struct(
      int              count                          /* in  */,
      int              array_of_blocklengths[]        /* in  */,
      MPI_Aint         array_of_displacements[]       /* in  */,
      MPI_Datatype     array_of_types[]               /* in  */,
      MPI_Datatype*    new_type_p                     /* out */);
```

The argument `count` is the number of elements in the datatype, so for our example, it should be three. Each of the array arguments should have `count` elements. The first array, `array_of_block_lengths`, allows for the possibility that the individual data items might be arrays or subarrays. If, for example, the first element were an array containing five elements, we would have

```
array_of_blocklengths[0] = 5;
```

However, in our case, none of the elements is an array, so we can simply define

```
int array_of_blocklengths[3] = {1, 1, 1};
```

The third argument to `MPI_Type_create_struct`, `array_of_displacements` specifies the displacements in bytes, from the start of the message. So we want

```
array_of_displacements[] = {0, 16, 24};
```

To find these values, we can use the function `MPI_Get_address`:

```
int MPI_Get_address(
      void*       location_p   /* in  */,
      MPI_Aint*   address_p    /* out */);
```

It returns the address of the memory location referenced by `location_p`. The special type `MPI_Aint` is an integer type that is big enough to store an address on the system. Thus to get the values in `array_of_displacements`, we can use the following code:

```
MPI_Aint a_addr, b_addr, n_addr;

MPI_Get_address(&a, &a_addr);
array_of_displacements[0] = 0;
MPI_Get_address(&b, &b_addr);
array_of_displacements[1] = b_addr - a_addr;
MPI_Get_address(&n, &n_addr);
array_of_displacements[2] = n_addr - a_addr;
```

The `array_of_datatypes` should store the MPI datatypes of the elements. So we can just define

```
MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE,
      MPI_INT};
```

With these initializations, we can build the new datatype with the call

```
MPI_Datatype input_mpi_t;
. . .
MPI_Type_create_struct(3, array_of_blocklengths,
      array_of_displacements, array_of_types,
      &input_mpi_t);
```

Before we can use `input_mpi_t` in a communication function, we must first **commit** it with a call to

```
int MPI_Type_commit(
      MPI_Datatype*  new_mpi_t_p  /* in/out */);
```

This allows the MPI implementation to optimize its internal representation of the datatype for use in communication functions.

Now, to use `input_mpi_t`, we make the following call to `MPI_Bcast` on each process:

```
MPI_Bcast(&a, 1, input_mpi_t, 0, comm);
```

So we can use `input_mpi_t`, just as we would use one of the basic MPI datatypes.

In constructing the new datatype, it's likely that the MPI implementation had to allocate additional storage internally. So when we're through using the new type, we can free any additional storage used with a call to

```
int MPI_Type_free(MPI_Datatype*  old_mpi_t_p  /* in/out */);
```

We used the steps outlined here to define a `Build_mpi_type` function that our `Get_input` function can call. The new function and the updated `Get_input` function are shown in Program 3.13.

3.6 Performance evaluation of MPI programs

Let's take a look at the performance of the matrix-vector multiplication program. For the most part, we write parallel programs because we expect that they'll be faster than a serial program that solves the same problem. How can we verify this? We spent some time discussing this in Section 2.6, so we'll start by recalling some of the material we learned there.

3.6.1 Taking timings

We're usually not interested in the time taken from the start of program execution to the end of program execution. For example, in the matrix-vector multiplication, we're not interested in the time it takes to type in the matrix or print out the product. We're only interested in the time it takes to do the actual multiplication, so we need to modify our source code by adding in calls to a function that will tell us the amount of time that elapses from the beginning to the end of the actual matrix-vector multiplication. MPI provides a function, `MPI_Wtime`, that returns the number of seconds that have elapsed since some time in the past:

```
double MPI_Wtime(void);
```

We can time a block of MPI code as follows:

```
double start, finish;
. . .
start = MPI_Wtime();
/* Code to be timed */
. . .
```

```
void Build_mpi_type(
      double*         a_p              /* in  */,
      double*         b_p              /* in  */,
      int*            n_p              /* in  */,
      MPI_Datatype*   input_mpi_t_p    /* out */) {

   int array_of_blocklengths[3] = {1, 1, 1};
   MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE,
      MPI_INT};
   MPI_Aint a_addr, b_addr, n_addr;
   MPI_Aint array_of_displacements[3] = {0};

   MPI_Get_address(a_p, &a_addr);
   MPI_Get_address(b_p, &b_addr);
   MPI_Get_address(n_p, &n_addr);
   array_of_displacements[1] = b_addr-a_addr;
   array_of_displacements[2] = n_addr-a_addr;
   MPI_Type_create_struct(3, array_of_blocklengths,
         array_of_displacements, array_of_types,
         input_mpi_t_p);
   MPI_Type_commit(input_mpi_t_p);
}  /* Build_mpi_type */

void Get_input(int my_rank, int comm_sz, double* a_p,
      double* b_p, int* n_p) {
   MPI_Datatype input_mpi_t;

   Build_mpi_type(a_p, b_p, n_p, &input_mpi_t);

   if (my_rank == 0) {
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);
   }
   MPI_Bcast(a_p, 1, input_mpi_t, 0, MPI_COMM_WORLD);

   MPI_Type_free(&input_mpi_t);
}  /* Get_input */
```

Program 3.13: The `Get_input` function with a derived datatype.

```
      finish = MPI_Wtime();
      printf("Proc %d > Elapsed time = %e seconds\n"
            my_rank, finish-start);
```

To time serial code, it's not necessary to link in the MPI libraries. There is a POSIX library function called `gettimeofday` that returns the number of microseconds

that have elapsed since some point in the past. The syntax details aren't too important: there's a C macro `GET_TIME` defined in the header file `timer.h` that can be downloaded from the book's website. This macro should be called with a double argument:

```
#include "timer.h"
. . .
double now;
. . .
GET_TIME(now);
```

After executing this macro, `now` will store the number of seconds since some time in the past. So we can get the elapsed time of serial code with microsecond resolution by executing

```
#include "timer.h"
. . .
double start, finish;
. . .
GET_TIME(start);
/* Code to be timed */
. . .
GET_TIME(finish);
printf("Elapsed time = %e seconds\n", finish-start);
```

One point to stress here: `GET_TIME` is a macro, so the code that defines it is inserted directly into your source code by the preprocessor. Hence, it can operate directly on its argument, and the argument is a **double**, *not* a pointer to a **double**. A final note in this connection: Since `timer.h` is not in the system include file directory, it's necessary to tell the compiler where to find it if it's not in the directory where you're compiling. For example, if it's in the directory `/home/peter/my_include`, the following command can be used to compile a serial program that uses `GET_TIME`:

```
$ gcc -g -Wall -I/home/peter/my_include -o <executable>
      <source_code.c>
```

Both `MPI_Wtime` and `GET_TIME` return *wall clock* time. Recall that a timer, such as the C `clock` function, returns CPU time: the time spent in user code, library functions, and operating system code. It doesn't include idle time, which can be a significant part of parallel run time. For example, a call to `MPI_Recv` may spend a significant amount of time waiting for the arrival of a message. Wall clock time, on the other hand, gives total elapsed time. So it includes idle time.

There are still a few remaining issues. First, as we've described it, our parallel program will report `comm_sz` times: one for each process. We would like to have it report a single time. Ideally, all of the processes would start execution of the matrix-vector multiplication at the same time, and then we would report the time that elapsed when the last process finished. In other words, the parallel execution time would be the time it took the "slowest" process to finish. We can't get exactly this time because we can't ensure that all the processes start at the same instant. However, we can come reasonably close. The MPI collective communication function `MPI_Barrier` ensures

that no process will return from calling it until every process in the communicator has started calling it. Its syntax is

```
int MPI_Barrier(MPI_Comm    comm    /* in */);
```

So the following code can be used to time a block of MPI code and report a single elapsed time:

```
double local_start, local_finish, local_elapsed, elapsed;
. . .
MPI_Barrier(comm);
local_start = MPI_Wtime();
/* Code to be timed */
. . .
local_finish = MPI_Wtime();
local_elapsed = local_finish - local_start;
MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE,
   MPI_MAX, 0, comm);

if (my_rank == 0)
   printf("Elapsed time = %e seconds\n", elapsed);
```

Note that the call to `MPI_Reduce` is using the `MPI_MAX` operator: it finds the largest of the input arguments `local_elapsed`.

As we noted in Chapter 2, we also need to be aware of variability in timings: when we run a program several times, we're likely to see a substantial variation in the times. This will be true, even if for each run we use the same input, the same number of processes, and the same system. This is because the interaction of the program with the rest of the system, especially the operating system, is unpredictable. Since this interaction will almost certainly not make the program run faster than it would run on a "quiet" system, we usually report the *minimum* run-time, rather than the mean or median. (For further discussion of this, see [6].)

Finally, when we run an MPI program on a hybrid system in which the nodes are multicore processors, we'll only run one MPI process on each node. This may reduce contention for the interconnect and result in somewhat better run-times. It may also reduce variability in run-times.

3.6.2 Results

The results of timing the matrix-vector multiplication program are shown in Table 3.5. The input matrices were square. The times shown are in milliseconds, and we've rounded each time to two significant digits. The times for `comm_sz` = 1 are the run-times of the serial program running on a single core of the distributed memory system. Not surprisingly, if we fix `comm_sz`, and increase n, the order of the matrix, the run-times increase. For relatively small numbers of processes, doubling n results in roughly a four-fold increase in the run-time. However, for large numbers of processes, this formula breaks down.

Table 3.5 Run-times of serial and parallel matrix-vector multiplication (times are in milliseconds).

	Order of Matrix				
comm_sz	1024	2048	4096	8192	16,384
1	4.1	16.0	64.0	270	1100
2	2.3	8.5	33.0	140	560
4	2.0	5.1	18.0	70	280
8	1.7	3.3	9.8	36	140
16	1.7	2.6	5.9	19	71

If we fix n and increase `comm_sz`, the run-times usually decrease. In fact, for large values of n, doubling the number of processes roughly halves the overall run-time. However, for small n, there is very little benefit in increasing `comm_sz`. In fact, in going from 8 to 16 processes when $n = 1024$, the overall run time is unchanged.

These timings are fairly typical of parallel run-times—as we increase the problem size, the run-times increase, and this is true regardless of the number of processes. The rate of increase can be fairly constant (e.g., the one process times) or it can vary wildly (e.g., the sixteen process times). As we increase the number of processes, the run-times typically decrease for a while. However, at some point, the run-times can actually start to get worse. The closest we came to this behavior was going from 8 to 16 processes when the matrix had order 1024.

The explanation for this is that there is a fairly common relation between the run-times of serial programs and the run-times of corresponding parallel programs. Recall that we denote the serial run-time by T_{serial}. Since it typically depends on the size of the input, n, we'll frequently denote it as $T_{\text{serial}}(n)$. Also recall that we denote the parallel run-time by T_{parallel}. Since it depends on both the input size, n, and the number of processes, `comm_sz` $= p$, we'll frequently denote it as $T_{\text{parallel}}(n, p)$. As we noted in Chapter 2, it's often the case that the parallel program will divide the work of the serial program among the processes, and add in some overhead time, which we denoted T_{overhead}:

$$T_{\text{parallel}}(n, p) = T_{\text{serial}}(n)/p + T_{\text{overhead}}.$$

In MPI programs, the parallel overhead typically comes from communication, and it can depend on both the problem size and the number of processes.

It's not to hard to see that this formula applies to our matrix-vector multiplication program. The heart of the serial program is the pair of nested for loops:

```
for (i = 0; i < m; i++) {
   y[i] = 0.0;
   for (j = 0; j < n; j++)
      y[i] += A[i*n+j]*x[j];
}
```

If we only count floating point operations, the inner loop carries out n multiplications and n additions, for a total of $2n$ floating point operations. Since we execute the inner loop m times, the pair of loops executes a total of $2mn$ floating point operations. So when $m = n$,

$$T_{\text{serial}}(n) \approx an^2$$

for some constant a. (The symbol $\approx$ means "is approximately equal to.")

If the serial program multiplies an $n \times n$ matrix by an n-dimensional vector, then each process in the parallel program multiplies an $n/p \times n$ matrix by an n-dimensional vector. The *local* matrix-vector multiplication part of the parallel program therefore executes n^2/p floating point operations. Thus it appears that this *local* matrix-vector multiplication reduces the work per process by a factor of p.

However, the parallel program also needs to complete a call to `MPI_Allgather` before it can carry out the local matrix-vector multiplication. In our example, it appears that

$$T_{\text{parallel}}(n, p) = T_{\text{serial}}(n)/p + T_{\text{allgather}}.$$

Furthermore, in light of our timing data, it appears that for smaller values of p and larger values of n, the dominant term in our formula is $T_{\text{serial}}(n)/p$. To see this, observe first that for small p (e.g., $p = 2, 4$), doubling p roughly halves the overall run time. For example,

$$\begin{aligned}
T_{\text{serial}}(4096) &= 1.9 \times T_{\text{parallel}}(4096, 2)\\
T_{\text{serial}}(8192) &= 1.9 \times T_{\text{parallel}}(8192, 2)\\
T_{\text{parallel}}(8192, 2) &= 2.0 \times T_{\text{parallel}}(8192, 4)\\
T_{\text{serial}}(16{,}384) &= 2.0 \times T_{\text{parallel}}(16{,}384, 2)\\
T_{\text{parallel}}(16{,}384, 2) &= 2.0 \times T_{\text{parallel}}(16{,}384, 4)
\end{aligned}$$

Also, if we fix p at a small value (e.g., $p = 2, 4$), then increasing n seems to have approximately the same effect as increasing n for the serial program. For example,

$$\begin{aligned}
T_{\text{serial}}(4096) &= 4.0 \times T_{\text{serial}}(2048)\\
T_{\text{parallel}}(4096, 2) &= 3.9 \times T_{\text{serial}}(2048, 2)\\
T_{\text{parallel}}(4096, 4) &= 3.5 \times T_{\text{serial}}(2048, 4)\\
T_{\text{serial}}(8192) &= 4.2 \times T_{\text{serial}}(4096)\\
T_{\text{serial}}(8192, 2) &= 4.2 \times T_{\text{parallel}}(4096, 2)\\
T_{\text{parallel}}(8192, 4) &= 3.9 \times T_{\text{parallel}}(8192, 4)
\end{aligned}$$

These observations suggest that the parallel run-times are behaving much as the run-times of the serial program, i.e., $T_{\text{serial}}(n)/p$. In other words, the overhead $T_{\text{allgather}}$ has little effect on the performance.

Table 3.6 Speedups of parallel matrix-vector multiplication.

	Order of Matrix				
comm_sz	1024	2048	4096	8192	16,384
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

On the other hand, for small n and large p, these patterns break down. For example,

$$T_{\text{parallel}}(1024, 8) = 1.0 \times T_{\text{parallel}}(1024, 16)$$
$$T_{\text{parallel}}(2048, 16) = 1.5 \times T_{\text{parallel}}(1024, 16)$$

So it appears that for small n and large p, the dominant term in our formula for T_{parallel} is $T_{\text{allgather}}$.

3.6.3 Speedup and efficiency

Recall that the most widely used measure of the relation between the serial and the parallel run-times is the **speedup**. It's just the ratio of the serial run-time to the parallel run-time:

$$S(n, p) = \frac{T_{\text{serial}}(n)}{T_{\text{parallel}}(n, p)}.$$

The ideal value for $S(n, p)$ is p. If $S(n, p) = p$, then our parallel program with comm_sz $= p$ processes is running p times faster than the serial program. In practice, this speedup, sometimes called **linear speedup**, is not often achieved. Our matrix-vector multiplication program got the speedups shown in Table 3.6. For small p and large n, our program obtained nearly linear speedup. On the other hand, for large p and small n, the speedup was considerably less than p. The worst case being $n = 1024$ and $p = 16$, when we only managed a speedup of 2.4.

Also recall that another widely used measure of parallel performance is parallel **efficiency**. This is "per process" speedup:

$$E(n, p) = \frac{S(n, p)}{p} = \frac{T_{\text{serial}}(n)}{p \times T_{\text{parallel}}(n, p)}.$$

So linear speedup corresponds to a parallel efficiency of $p/p = 1.0$, and, in general, we expect that our efficiencies will usually be less than 1.

Table 3.7 Efficiencies of parallel matrix-vector multiplication.

	Order of Matrix				
comm_sz	1024	2048	4096	8192	16,384
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

The efficiencies for the matrix-vector multiplication program are shown in Table 3.7. Once again, for small p and large n, our parallel efficiencies are near linear, and for large p and small n, they are very far from linear.

3.6.4 Scalability

Our parallel matrix-vector multiplication program doesn't come close to obtaining linear speedup for small n and large p. Does this mean that it's not a good program? Many computer scientists answer this question by looking at the "scalability" of the program. Recall that very roughly speaking a program is **scalable** if the problem size can be increased at a rate so that the efficiency doesn't decrease as the number of processes increase.

The problem with this definition is the phrase "the problem size can be increased at a rate ...." Consider two parallel programs: program A and program B. Suppose that if $p \geq 2$, the efficiency of program A is 0.75, regardless of problem size. Also suppose that the efficiency of program B is $n/(625p)$, provided $p \geq 2$ and $1000 \leq n \leq 625p$. Then according to our "definition," both programs are scalable. For program A, the rate of increase needed to maintain constant efficiency is 0, while for program B if we increase n at the same rate as we increase p, we'll maintain a constant efficiency. For example, if $n = 1000$ and $p = 2$, the efficiency of B is 0.80. If we then double p to 4 and we leave the problem size at $n = 1000$, the efficiency will drop to 0.4, but if we also double the problem size to $n = 2000$, the efficiency will remain constant at 0.8. Program A is thus *more* scalable than B, but both satisfy our definition of scalability.

Looking at our table of parallel efficiencies (Table 3.7), we see that our matrix-vector multiplication program definitely doesn't have the same scalability as program A: in almost every case when p is increased, the efficiency decreases. On the other hand, the program is somewhat like program B: if $p \geq 2$ and we increase both p and n by a factor of 2, the parallel efficiency, for the most part, actually increases. Furthermore, the only exceptions occur when we increase p from 2 to 4, and when computer scientists discuss scalability, they're usually interested in large values of p. When p is increased from 4 to 8 or from 8 to 16, our efficiency always increases when we increase n by a factor of 2.

Recall that programs that can maintain a constant efficiency without increasing the problem size are sometimes said to be **strongly scalable**. Programs that can maintain a constant efficiency if the problem size increases at the same rate as the number of processes are sometimes said to be **weakly scalable**. Program A is strongly scalable, and program B is weakly scalable. Furthermore, our matrix-vector multiplication program is also apparently weakly scalable.

3.7 A parallel sorting algorithm

What do we mean by a parallel sorting algorithm in a distributed memory environment? What would its "input" be and what would its "output" be? The answers depend on where the keys are stored. We can start or finish with the keys distributed among the processes or assigned to a single process. In this section, we'll look at an algorithm that starts and finishes with the keys distributed among the processes. In programming assignment 3.8, we'll look at an algorithm that finishes with the keys assigned to a single process.

If we have a total of n keys and $p =$ `comm_sz` processes, our algorithm will start and finish with n/p keys assigned to each process. (As usual, we'll assume n is evenly divisible by p.) At the start, there are no restrictions on which keys are assigned to which processes. However, when the algorithm terminates,

- the keys assigned to each process should be sorted in (say) increasing order, and
- if $0 \leq q < r < p$, then each key assigned to process q should be less than or equal to every key assigned to process r.

So if we lined up the keys according to process rank—keys from process 0 first, then keys from process 1, and so on—then the keys would be sorted in increasing order. For the sake of explicitness, we'll assume our keys are ordinary **ints**.

3.7.1 Some simple serial sorting algorithms

Before starting, let's look at a couple of simple serial sorting algorithms. Perhaps the best-known serial sorting algorithm is bubble sort. (See Program 3.14.) The array `a` stores the unsorted keys when the function is called, and the sorted keys when the function returns. The number of keys in `a` is `n`. The algorithm proceeds by comparing the elements of the list `a` pairwise: `a[0]` is compared to `a[1]`, `a[1]` is compared to `a[2]`, and so on. Whenever a pair is out of order, the entries are swapped, so in the first pass through the outer loop, when `list_length` = `n`, the largest value in the list will be moved into `a[n-1]`. The next pass will ignore this last element, and it will move the next-to-the-largest element into `a[n-2]`. Thus as `list_length` decreases, successively more elements get assigned to their final positions in the sorted list.

There isn't much point in trying to parallelize this algorithm because of the inherently sequential ordering of the comparisons. To see this, suppose that `a[i-1]` = 9, `a[i]` = 5, and `a[i+1]` = 7. The algorithm will first compare 9 and 5 and swap them;

```
void Bubble_sort(
      int  a[]  /* in/out */,
      int  n    /* in     */) {
   int list_length, i, temp;

   for (list_length = n; list_length >= 2; list_length--)
      for (i = 0; i < list_length-1; i++)
         if (a[i] > a[i+1]) {
            temp = a[i];
            a[i] = a[i+1];
            a[i+1] = temp;
         }

}  /* Bubble_sort */
```

Program 3.14: Serial bubble sort.

it will then compare 9 and 7 and swap them; and we'll have the sequence 5, 7, 9. If we try to do the comparisons out of order, that is, if we compare the 5 and 7 first and then compare the 9 and 5, we'll wind up with the sequence 5, 9, 7. Therefore the order in which the "compare-swaps" take place is essential to the correctness of the algorithm.

A variant of bubble sort, known as *odd-even transposition sort*, has considerably more opportunities for parallelism. The key idea is to "decouple" the compare-swaps. The algorithm consists of a sequence of *phases*, of two different types. During *even* phases, compare-swaps are executed on the pairs

$$(a[0], a[1]), (a[2], a[3]), (a[4], a[5]), \ldots,$$

and during *odd* phases, compare-swaps are executed on the pairs

$$(a[1], a[2]), (a[3], a[4]), (a[5], a[6]), \ldots.$$

Here's a small example:

Start: 5, 9, 4, 3
Even phase: Compare-swap (5, 9) and (4, 3), getting the list 5, 9, 3, 4.
Odd phase: Compare-swap (9, 3), getting the list 5, 3, 9, 4.
Even phase: Compare-swap (5, 3), and (9, 4) getting the list 3, 5, 4, 9.
Odd phase: Compare-swap (5, 4) getting the list 3, 4, 5, 9.

This example required four phases to sort a four-element list. In general, it may require fewer phases, but the following theorem guarantees that we can sort a list of n elements in at most n phases.

Theorem. *Suppose A is a list with n keys, and A is the input to the odd-even transposition sort algorithm. Then after n phases, A will be sorted.*

Program 3.15 shows code for a serial odd-even transposition sort function.

```
void Odd_even_sort(
      int  a[]  /* in/out */,
      int  n    /* in     */) {
   int phase, i, temp;

   for (phase = 0; phase < n; phase++)
      if (phase % 2 == 0) { /* Even phase */
         for (i = 1; i < n; i += 2)
            if (a[i-1] > a[i]) {
               temp = a[i];
               a[i] = a[i-1];
               a[i-1] = temp;
            }
      } else { /* Odd phase */
         for (i = 1; i < n-1; i += 2)
            if (a[i] > a[i+1]) {
               temp = a[i];
               a[i] = a[i+1];
               a[i+1] = temp;
            }
      }
}  /* Odd_even_sort */
```

Program 3.15: Serial odd-even transposition sort.

3.7.2 Parallel odd-even transposition sort

It should be clear that odd-even transposition sort has considerably more opportunities for parallelism than bubble sort, because all of the compare-swaps in a single phase can happen simultaneously. Let's try to exploit this.

There are a number of possible ways to apply Foster's methodology. Here's one:

- *Tasks*: Determine the value of `a[i]` at the end of phase j.
- *Communications*: The task that's determining the value of `a[i]` needs to communicate with either the task determining the value of `a[i-1]` or `a[i+1]`. Also the value of `a[i]` at the end of phase j needs to be available for determining the value of `a[i]` at the end of phase $j+1$.

This is illustrated in Fig. 3.12, where we've labeled the task determining the value of `a[k]` with `a[k]`.

Now recall that when our sorting algorithm starts and finishes execution, each process is assigned n/p keys. In this case, our aggregation and mapping are at least partially specified by the description of the problem. Let's look at two cases.

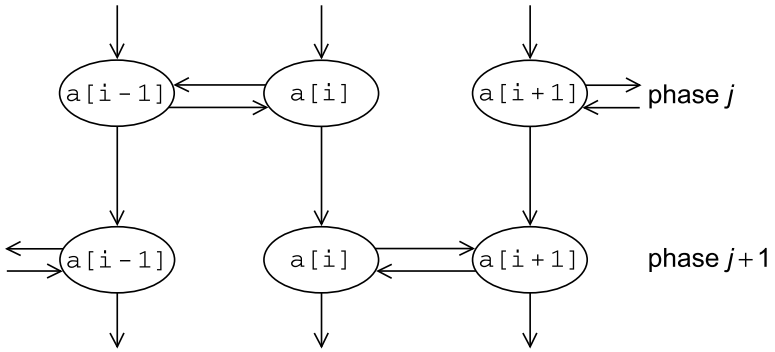

FIGURE 3.12

Communications among tasks in odd-even sort. Tasks determining `a[k]` are labeled with `a[k]`.

When $n = p$, Fig. 3.12 makes it fairly clear how the algorithm should proceed. Depending on the phase, process i can send its current value, `a[i]`, either to process $i - 1$ or process $i + 1$. At the same time, it should receive the value stored on process $i - 1$ or process $i + 1$, respectively, and then decide which of the two values it should store as `a[i]` for the next phase.

However, it's unlikely that we'll actually want to apply the algorithm when $n = p$, since we're unlikely to have more than a few hundred or a few thousand processors at our disposal, and sorting a few thousand values is usually a fairly trivial matter for a single processor. Furthermore, even if we do have access to thousands or even millions of processors, the added cost of sending and receiving a message for each compare-exchange will slow the program down so much that it will be useless. Remember that the cost of communication is usually much greater than the cost of "local" computation—for example, a compare-swap.

How should this be modified when each process is storing $n/p > 1$ elements? (Recall that we're assuming that n is evenly divisible by p.) Let's look at an example. Suppose we have $p = 4$ processes and $n = 16$ keys, as shown in Table 3.8. In the first place, we can apply a fast serial sorting algorithm to the keys assigned to each process. For example, we can use the C library function `qsort` on each process to sort the local keys. Now if we had one element per process, 0 and 1 would exchange elements, and 2 and 3 would exchange. So let's try this: Let's have 0 and 1 exchange *all* their elements, and 2 and 3 exchange all of theirs. Then it would seem natural for 0 to keep the four smaller elements and 1 to keep the larger. Similarly, 2 should keep the smaller and 3 the larger. This gives us the situation shown in the third row of the table. Once again, looking at the one element per process case, in phase 1, processes 1 and 2 exchange their elements and processes 0 and 4 are idle. If process 1 keeps the smaller and 2 the larger elements, we get the distribution shown in the fourth row. Continuing this process for two more phases results in a sorted list. That is, each

Table 3.8 Parallel odd-even transposition sort.

	Process			
Time	**0**	**1**	**2**	**3**
Start	15, 11, 9, 16	3, 14, 8, 7	4, 6, 12, 10	5, 2, 13, 1
After Local Sort	9, 11, 15, 16	3, 7, 8, 14	4, 6, 10, 12	1, 2, 5, 13
After Phase 0	3, 7, 8, 9	11, 14, 15, 16	1, 2, 4, 5	6, 10, 12, 13
After Phase 1	3, 7, 8, 9	1, 2, 4, 5	11, 14, 15, 16	6, 10, 12, 13
After Phase 2	1, 2, 3, 4	5, 7, 8, 9	6, 10, 11, 12	13, 14, 15, 16
After Phase 3	1, 2, 3, 4	5, 6, 7, 8	9, 10, 11, 12	13, 14, 15, 16

process's keys are stored in increasing order, and if $q < r$, then the keys assigned to process q are less than or equal to the keys assigned to process r.

In fact, our example illustrates the worst-case performance of this algorithm:

Theorem. *If parallel odd-even transposition sort is run with p processes, then after p phases, the input list will be sorted.*

The parallel algorithm is clear to a human computer:

```
Sort local keys;
for (phase = 0; phase < comm_sz; phase++) {
   partner = Compute_partner(phase, my_rank);
   if (I'm not idle) {
      Send my keys to partner;
      Receive keys from partner;
      if (my_rank < partner)
         Keep smaller keys;
      else
         Keep larger keys;
   }
}
```

However, there are some details that we need to clear up before we can convert the algorithm into an MPI program.

First, how do we compute the partner rank? And what is the partner rank when a process is idle? If the phase is even, then odd-ranked partners exchange with `my_rank` -1 and even-ranked partners exchange with `my_rank`+1. In odd phases, the calculations are reversed. However, these calculations can return some invalid ranks: if `my_rank` = 0 or `my_rank` = `comm_sz`-1, the partner rank can be -1 or `comm_sz`. But when either `partner` = -1 or `partner` = `comm_sz`, the process should be idle. So we can use the rank computed by `Compute_partner` to determine whether a process is idle:

```
if (phase % 2 == 0)          /* Even phase */
   if (my_rank % 2 != 0)     /* Odd rank */
      partner = my_rank - 1;
   else                      /* Even rank */
      partner = my_rank + 1;
```

```
else                        /* Odd phase */
   if (my_rank % 2 != 0)      /* Odd rank */
      partner = my_rank + 1;
   else                       /* Even rank */
      partner = my_rank - 1;
if (partner == -1 || partner == comm_sz)
   partner = MPI_PROC_NULL;
```

MPI_PROC_NULL is a constant defined by MPI. When it's used as the source or destination rank in a point-to-point communication, no communication will take place, and the call to the communication will simply return.

3.7.3 Safety in MPI programs

If a process is not idle, we might try to implement the communication with a call to MPI_Send and a call to MPI_Recv:

```
MPI_Send(my_keys, n/comm_sz, MPI_INT, partner, 0, comm);
MPI_Recv(temp_keys, n/comm_sz, MPI_INT, partner, 0, comm,
      MPI_STATUS_IGNORE);
```

This, however, might result in the program's hanging or crashing. Recall that the MPI standard allows MPI_Send to behave in two different ways: it can simply copy the message into an MPI-managed buffer and return, or it can block until the matching call to MPI_Recv starts. Furthermore, many implementations of MPI set a threshold at which the system switches from buffering to blocking. That is, messages that are relatively small will be buffered by MPI_Send, but for larger messages, it will block. If the MPI_Send executed by each process blocks, no process will be able to start executing a call to MPI_Recv, and the program will hang or **deadlock**: that is, each process is blocked waiting for an event that will never happen.

A program that relies on MPI-provided buffering is said to be **unsafe**. Such a program may run without problems for various sets of input, but it may hang or crash with other sets. If we use MPI_Send and MPI_Recv in this way, our program will be unsafe, and it's likely that for small values of n the program will run without problems, while for larger values of n, it's likely that it will hang or crash.

There are a couple of questions that arise here:

1. In general, how can we tell if a program is safe?
2. How can we modify the communication in the parallel odd-even sort program so that it is safe?

To answer the first question, we can use an alternative to MPI_Send defined by the MPI standard. It's called MPI_Ssend. The extra "s" stands for *synchronous*, and MPI_Ssend is guaranteed to block until the matching receive starts. So, we can check whether a program is safe by replacing the calls to MPI_Send with calls to MPI_Ssend. If the program doesn't hang or crash when it's run with appropriate input and comm_sz, then the original program was safe. The arguments to MPI_Ssend are the same as the arguments to MPI_Send:

```
int MPI_Ssend(
      void*          msg_buf_p      /* in */,
      int            msg_size       /* in */,
      MPI_Datatype   msg_type       /* in */,
      int            dest           /* in */,
      int            tag            /* in */,
      MPI_Comm       communicator   /* in */);
```

The answer to the second question is that the communication must be restructured. The most common cause of an unsafe program is multiple processes simultaneously first sending to each other and then receiving. Our exchanges with partners is one example. Another example is a "ring pass," in which each process q sends to the process with rank $q + 1$, except that process comm_sz $- 1$ sends to 0:

```
MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0,
      comm);
MPI_Recv(new_msg, size, MPI_INT,
      (my_rank+comm_sz-1) % comm_sz, 0, comm,
      MPI_STATUS_IGNORE);
```

In both settings, we need to restructure the communications so that some of the processes receive before sending. For example, the preceding communications could be restructured as follows:

```
if (my_rank % 2 == 0) {
   MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0,
         comm);
   MPI_Recv(new_msg, size, MPI_INT,
         (my_rank+comm_sz-1) % comm_sz, 0, comm,
         MPI_STATUS_IGNORE);
} else {
   MPI_Recv(new_msg, size, MPI_INT,
         (my_rank+comm_sz-1) % comm_sz, 0, comm,
         MPI_STATUS_IGNORE);
   MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0,
         comm);
}
```

It's fairly clear that this will work if comm_sz is even. If, say, comm_sz $= 4$, then processes 0 and 2 will first send to 1 and 3, respectively, while processes 1 and 3 will receive from 0 and 2, respectively. The roles are reversed for the next send-receive pairs: processes 1 and 3 will send to 2 and 0, respectively, while 2 and 0 will receive from 1 and 3.

However, it may not be clear that this scheme is also safe if comm_sz is odd (and greater than 1). Suppose, for example, that comm_sz $= 5$. Then Fig. 3.13 shows a possible sequence of events. The solid arrows show a completed communication, and the dashed arrows show a communication waiting to complete.

MPI provides an alternative to scheduling the communications ourselves—we can call the function MPI_Sendrecv:

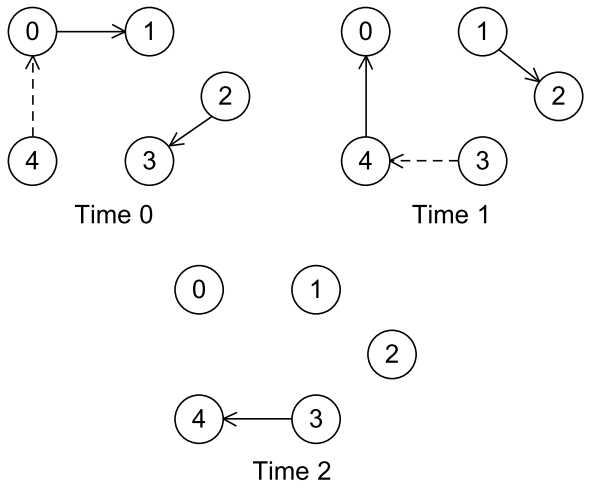

FIGURE 3.13

Safe communication with five processes.

```
int MPI_Sendrecv(
      void*          send_buf_p      /* in  */,
      int            send_buf_size   /* in  */,
      MPI_Datatype   send_buf_type   /* in  */,
      int            dest            /* in  */,
      int            send_tag        /* in  */,
      void*          recv_buf_p      /* out */,
      int            recv_buf_size   /* in  */,
      MPI_Datatype   recv_buf_type   /* in  */,
      int            source          /* in  */,
      int            recv_tag        /* in  */,
      MPI_Comm       communicator    /* in  */,
      MPI_Status*    status_p        /* in  */);
```

This function carries out a blocking send and a receive in a single call. The `dest` and the `source` can be the same or different. What makes it especially useful is that the MPI implementation schedules the communications so that the program won't hang or crash. The complex code we used above—the code that checks whether the process rank is odd or even—can be replaced with a single call to `MPI_Sendrecv`. If it happens that the send and the receive buffers should be the same, MPI provides the alternative:

```
int MPI_Sendrecv_replace(
      void*          buf_p           /* in/out */,
      int            buf_size        /* in     */,
      MPI_Datatype   buf_type        /* in     */,
      int            dest            /* in     */,
```

```
      int           send_tag       /* in  */,
      int           source         /* in  */,
      int           recv_tag       /* in  */,
      MPI_Comm      communicator   /* in  */,
      MPI_Status*   status_p       /* in  */);
```

3.7.4 Final details of parallel odd-even sort

Recall that we had developed the following parallel odd-even transposition sort algorithm:

```
Sort local keys;
for (phase = 0; phase < comm_sz; phase++) {
   partner = Compute_partner(phase, my_rank);
   if (I'm not idle) {
      Send my keys to partner;
      Receive keys from partner;
      if (my_rank < partner)
         Keep smaller keys;
      else
         Keep larger keys;
   }
}
```

In light of our discussion of safety in MPI, it probably makes sense to implement the send and the receive with a single call to `MPI_Sendrecv`:

```
MPI_Sendrecv(my_keys, n/comm_sz, MPI_INT, partner, 0,
      recv_keys, n/comm_sz, MPI_INT, partner, 0, comm,
      MPI_Status_ignore);
```

So it only remains to identify which keys we keep. Suppose for the moment that we want to keep the smaller keys. Then we want to keep the smallest n/p keys in a collection of $2n/p$ keys. An obvious approach to doing this is to sort (using a serial sorting algorithm) the list of $2n/p$ keys, and keep the first half of the list. However, sorting is a relatively expensive operation, and we can exploit the fact that we already have two sorted lists of n/p keys to reduce the cost by *merging* the two lists into a single list. In fact, we can do even better: we don't need a fully general merge; once we've found the smallest n/p keys, we can quit. See Program 3.16.

To get the largest n/p keys, we simply reverse the order of the merge. That is, we start with `local_n`-1 and work backward through the arrays. A final improvement avoids copying the arrays; it simply swaps pointers. (See Exercise 3.28.)

Run-times for the version of parallel odd-even sort with the "final improvement" are shown in Table 3.9. Note that if parallel odd-even sort is run on a single processor, it will use whatever serial sorting algorithm we use to sort the local keys. So the times for a single process use serial quicksort, not serial odd-even sort, which would be *much* slower. We'll take a closer look at these times in Exercise 3.27.

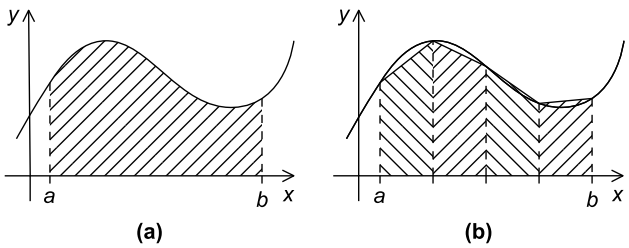

FIGURE 5.3

The trapezoidal rule.

Here, if OpenMP isn't available, we assume that the `Hello` function will be single-threaded. Thus the single thread's rank will be 0, and the number of threads will be 1.

The book's website contains the source for a version of this program that makes these checks. To make our code as clear as possible, we'll usually show little, if any, error checking in the code displayed in the text. We'll also assume that OpenMP is available and supported by the compiler.

5.2 The trapezoidal rule

Let's take a look at a somewhat more useful (and more complicated) example: the trapezoidal rule for estimating the area under a curve. Recall from Section 3.2 that if $y = f(x)$ is a reasonably nice function, and $a < b$ are real numbers, then we can estimate the area between the graph of $f(x)$, the vertical lines $x = a$ and $x = b$, and the x-axis by dividing the interval $[a, b]$ into n subintervals and approximating the area over each subinterval by the area of a trapezoid. See Fig. 5.3.

Also recall that if each subinterval has the same length and if we define $h = (b - a)/n$, $x_i = a + ih$, $i = 0, 1, \ldots, n$, then our approximation will be

$$h[\, f(x_0)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + f(x_n)/2].$$

Thus we can implement a serial algorithm using the following code:

```
/* Input:  a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++) {
   x_i = a + i*h;
   approx += f(x_i);
}
approx = h*approx;
```

See Section 3.2.1 for details.

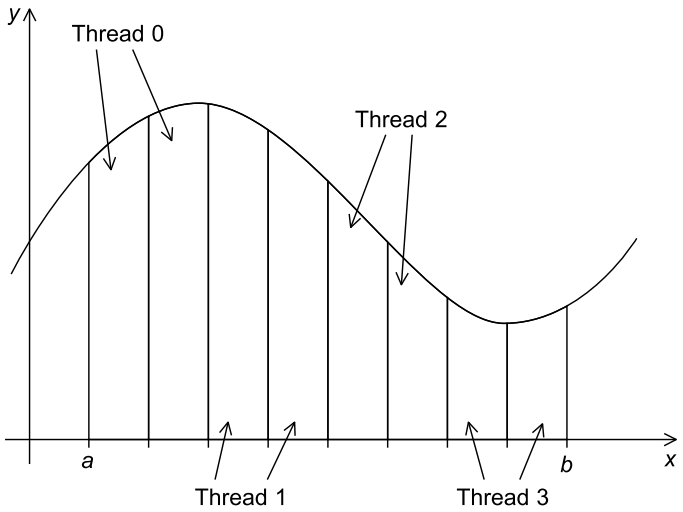

FIGURE 5.4

Assignment of trapezoids to threads.

5.2.1 A first OpenMP version

Recall that we applied Foster's parallel program design methodology to the trapezoidal rule as described in the following list (see Section 3.2.2):

1. We identified two types of jobs:
   a. Computation of the areas of individual trapezoids, and
   b. Adding the areas of trapezoids.
2. There is no communication among the jobs in the first collection, but each job in the first collection communicates with job 1b.
3. We assumed that there would be many more trapezoids than cores, so we aggregated jobs by assigning a contiguous block of trapezoids to each thread (and a single thread to each core).[2] Effectively, this partitioned the interval $[a, b]$ into larger subintervals, and each thread simply applied the serial trapezoidal rule to its subinterval. See Fig. 5.4.

We aren't quite done, however, since we still need to add up the threads' results. An obvious solution is to use a shared variable for the sum of all the threads' results, and each thread can add its (private) result into the shared variable. We would like to have each thread execute a statement that looks something like

```
global_result += my_result;
```

[2] Since we were discussing MPI, we actually used *processes* instead of threads.

However, as we've already seen, this can result in an erroneous value for `global_result`—if two (or more) threads attempt to simultaneously execute this statement, the result will be unpredictable. For example, suppose that `global_result` has been initialized to 0, thread 0 has computed `my_result` = 1, and thread 1 has computed `my_result` = 2. Furthermore, suppose that the threads execute the statement `global_result` += `my_result` according to the following timetable:

Time	Thread 0	Thread 1
0	`global_result = 0` to register	finish `my_result`
1	`my_result = 1` to register	`global_result = 0` to register
2	add `my_result` to `global_result`	`my_result = 2` to register
3	store `global_result = 1`	add `my_result` to `global_result`
4		store `global_result = 2`

We see that the value computed by thread 0 (`my_result` = 1) is overwritten by thread 1.

Of course, the actual sequence of events might be different, but unless one thread finishes the computation `global_result` += `my_result` before the other starts, the result will be incorrect. Recall that this is an example of a **race condition**: multiple threads are attempting to access a shared resource, at least one of the accesses is an update, and the accesses can result in an error. Also recall that the code that causes the race condition, `global_result` += `my_result`, is called a **critical section**. A critical section is code executed by multiple threads that updates a shared resource, and the shared resource can only be updated by one thread at a time.

We therefore need some mechanism to make sure that once one thread has started executing `global_result` += `my_result`, no other thread can start executing this code until the first thread has finished. In Pthreads we used mutexes or semaphores. In OpenMP we can use the `critical` directive

```
# pragma omp critical
  global_result += my_result;
```

This directive tells the compiler that the system needs to arrange for the threads to have **mutually exclusive** access to the following structured block of code.[3] That is, only one thread can execute the following structured block at a time. The code for this version is shown in Program 5.2. We've omitted any error checking. We've also omitted code for the function $f(x)$.

In the `main` function, prior to Line 17, the code is single-threaded, and it simply gets the number of threads and the input (a, b, and n). In Line 17 the `parallel` directive specifies that the `Trap` function should be executed by `thread_count` threads. After

[3] You are likely used to seeing blocks preceded by a control flow statement (for example, **if**, **for**, **while**, and so on). As you'll soon see, this needn't always be the case; if we wanted to define a critical section that spanned the next two lines of code, we would simply enclose it in curly braces.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5  void Trap(double a, double b, int n, double* global_result_p);
6
7  int main(int argc, char* argv[]) {
8     /* We'll store our result in global_result: */
9     double  global_result = 0.0;
10    double  a, b;  /* Left and right endpoints   */
11    int     n;     /* Total number of trapezoids */
12    int     thread_count;
13
14    thread_count = strtol(argv[1], NULL, 10);
15    printf("Enter a, b, and n\n");
16    scanf("%lf %lf %d", &a, &b, &n);
17 #  pragma omp parallel num_threads(thread_count)
18    Trap(a, b, n, &global_result);
19
20    printf("With n = %d trapezoids, our estimate\n", n);
21    printf("of the integral from %f to %f = %.14e\n",
22       a, b, global_result);
23    return 0;
24 }  /* main */
25
26 void Trap(double a, double b, int n, double* global_result_p) {
27    double  h, x, my_result;
28    double  local_a, local_b;
29    int  i, local_n;
30    int my_rank = omp_get_thread_num();
31    int thread_count = omp_get_num_threads();
32
33    h = (b-a)/n;
34    local_n = n/thread_count;
35    local_a = a + my_rank*local_n*h;
36    local_b = local_a + local_n*h;
37    my_result = (f(local_a) + f(local_b))/2.0;
38    for (i = 1; i <= local_n-1; i++) {
39      x = local_a + i*h;
40      my_result += f(x);
41    }
42    my_result = my_result*h;
43
44 #  pragma omp critical
45    *global_result_p += my_result;
46 }  /* Trap */
```

Program 5.2: First OpenMP trapezoidal rule program.

returning from the call to `Trap`, any new threads that were started by the `parallel` directive are terminated, and the program resumes execution with only one thread. The one thread prints the result and terminates.

In the `Trap` function, each thread gets its rank and the total number of threads in the team started by the `parallel` directive. Then each thread determines the following:

1. The length of the bases of the trapezoids (Line 33),
2. The number of trapezoids assigned to each thread (Line 34),
3. The left and right endpoints of its interval (Lines 35 and 36, respectively)
4. Its contribution to `global_result` (Lines 37–42).

The threads finish by adding in their individual results to `global_result` in Lines 44–45.

We use the prefix `local_` for some variables to emphasize that their values may differ from the values of corresponding variables in the `main` function—for example, `local_a` may differ from `a`, although it is the *thread's* left endpoint.

Notice that unless n is evenly divisible by `thread_count`, we'll use fewer than n trapezoids for `global_result`. For example, if $n = 14$ and `thread_count` = 4, each thread will compute

```
local_n = n/thread_count = 14/4 = 3.
```

Thus each thread will only use 3 trapezoids, and `global_result` will be computed with $4 \times 3 = 12$ trapezoids instead of the requested 14. So in the error checking (which isn't shown), we check that n is evenly divisible by `thread_count` by doing something like this:

```
if (n % thread_count != 0) {
   fprintf(stderr,
      "n must be evenly divisible by thread_count\n");
   exit(0);
}
```

Since each thread is assigned a block of `local_n` trapezoids, the length of each thread's interval will be `local_n*h`, so the left endpoints will be

```
thread 0:  a + 0*local_n*h
thread 1:  a + 1*local_n*h
thread 2:  a + 2*local_h*h
   . . .
```

So in Line 35, we assign

```
local_a = a + my_rank*local_n*h;
```

Furthermore, since the length of each thread's interval will be `local_n*h`, its right endpoint will just be

```
local_b = local_a + local_n*h;
```

5.3 Scope of variables

In serial programming, the *scope* of a variable consists of those parts of a program in which the variable can be used. For example, a variable declared at the beginning of a C function has "function-wide" scope, that is, it can only be accessed in the body of the function. On the other hand, a variable declared at the beginning of a `.c` file but outside any function has "file-wide" scope, that is, any function in the file in which the variable is declared can access the variable. In OpenMP, the **scope** of a variable refers to the set of threads that can access the variable in a `parallel` block. A variable that can be accessed by all the threads in the team has **shared** scope, while a variable that can only be accessed by a single thread has **private** scope.

In the "hello, world" program, the variables used by each thread (`my_rank` and `thread_count`) were declared in the `Hello` function, which is called inside the `parallel` block. Consequently, the variables used by each thread are allocated from the thread's (private) stack, and hence all of the variables have private scope. This is *almost* the case in the trapezoidal rule program; since the `parallel` block is just a function call, all of the variables used by each thread in the `Trap` function are allocated from the thread's stack.

However, the variables that are declared in the `main` function (`a`, `b`, `n`, `global_result`, and `thread_count`) are all accessible to all the threads in the team started by the `parallel` directive. Hence, the *default* scope for variables declared before a `parallel` block is shared. In fact, we've made implicit use of this: each thread in the team gets the values of `a`, `b`, and `n` from the call to `Trap`. Since this call takes place in the `parallel` block, it's essential that each thread has access to `a`, `b`, and `n` when their values are copied into the corresponding formal arguments.

Furthermore, in the `Trap` function, although `global_result_p` is a private variable, it refers to the variable `global_result` which was declared in `main` before the `parallel` directive, and the value of `global_result` is used to store the result that's printed out after the `parallel` block. Thus in the code

```
*global_result_p += my_result;
```

it's essential that `*global_result_p` have shared scope. If it were private to each thread, there would be no need for the `critical` directive. Furthermore, if it were private, we would have a hard time determining the value of `global_result` in `main` after completion of the `parallel` block.

To summarize, then, variables that have been declared before a `parallel` directive have shared scope among the threads in the team, while variables declared in the block (e.g., local variables in functions) have private scope. Furthermore, the value of a shared variable at the beginning of the `parallel` block is the same as the value before the block, and, after completion of the `parallel` block, the value of the variable is the value at the end of the block.

We'll shortly see that the *default* scope of a variable can change with other directives, and that OpenMP provides clauses to modify the default scope.

5.4 The reduction clause

If we developed a serial implementation of the trapezoidal rule, we'd probably use a slightly different function prototype. Rather than

```
void Trap(
   double a,
   double b,
   int n,
   double* global_result_p);
```

we would probably define

```
double Trap(double a, double b, int n);
```

and our function call would be

```
global_result = Trap(a, b, n);
```

This is somewhat easier to understand and probably more attractive to all but the most fanatical believers in pointers.

We resorted to the pointer version, because we needed to add each thread's local calculation to get `global_result`. However, we might prefer the following function prototype:

```
double Local_trap(double a, double b, int n);
```

With this prototype, the body of `Local_trap` would be the same as the `Trap` function in Program 5.2, except that there would be no critical section. Rather, each thread would return its part of the calculation, the final value of its `my_result` variable. If we made this change, we might try modifying our `parallel` block so that it looks like this:

```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count)
   {
#     pragma omp critical
      global_result += Local_trap(double a, double b, int n);
   }
```

Can you see a problem with this code? It should give the correct result. However, since we've specified that the critical section is

```
global_result += Local_trap(double a, double b, int n);
```

the call to `Local_trap` can only be executed by one thread at a time, and, effectively, we're forcing the threads to execute the trapezoidal rule sequentially. If we check the run-time of this version, it may actually be *slower* with multiple threads than one thread (see Exercise 5.3).

We can avoid this problem by declaring a private variable inside the `parallel` block and moving the critical section after the function call:

```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count)
   {
      double my_result = 0.0;  /* private */
      my_result += Local_trap(double a, double b, int n);
#     pragma omp critical
      global_result += my_result;
   }
```

Now the call to `Local_trap` is outside the critical section, and the threads can execute their calls simultaneously. Furthermore, since `my_result` is declared in the `parallel` block, it's private, and before the critical section each thread will store its part of the calculation in its `my_result` variable.

OpenMP provides a cleaner alternative that also avoids serializing execution of `Local_trap`: we can specify that `global_result` is a *reduction* variable. A **reduction operator** is an associative binary operation (such as addition or multiplication), and a **reduction** is a computation that repeatedly applies the same reduction operator to a sequence of operands to get a single result. Furthermore, all of the intermediate results of the operation should be stored in the same variable: the **reduction variable**. For example, if `A` is an array of `n` **int**s, the computation

```
int sum = 0;
for (i = 0; i < n; i++)
   sum += A[i];
```

is a reduction in which the reduction operator is addition.

In OpenMP it may be possible to specify that the result of a reduction is a reduction variable. To do this, a `reduction` clause can be added to a `parallel` directive. In our example, we can modify the code as follows:

```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count) \
      reduction(+: global_result)
   global_result += Local_trap(double a, double b, int n);
```

First note that the `parallel` directive is two lines long. Recall that C preprocessor directives are, by default, only one line long, so we need to "escape" the newline character by putting a backslash (\) immediately before it.

The code specifies that `global_result` is a reduction variable, and the plus sign ("+") indicates that the reduction operator is addition. Effectively, OpenMP creates a private variable for each thread, and the run-time system stores each thread's result in this private variable. OpenMP also creates a critical section, and the values stored in the private variables are added in this critical section. Thus the calls to `Local_trap` can take place in parallel.

The syntax of the `reduction` clause is

```
reduction(<operator>: <variable list>)
```

In C, `operator` can be any one of the operators +, ∗, −, &, |, ^, &&, ‖. You may wonder whether the use of subtraction is problematic, though, since subtraction isn't

associative or commutative. For example, the serial code

```
result = 0;
for (i = 1; i <= 4; i++)
   result -= i;
```

stores the value -10 in `result`. However, if we split the iterations among two threads, with thread 0 subtracting 1 and 2 and thread 1 subtracting 3 and 4, then thread 0 will compute -3 and thread 1 will compute -7. This results in an incorrect calculation, $-3-(-7)=4$. Luckily, the OpenMP standard states that partial results of a subtraction reduction are *added* to form the final value, so the reduction will work as intended.

It should also be noted that if a reduction variable is a **float** or a **double**, the results may differ slightly when different numbers of threads are used. This is due to the fact that floating point arithmetic isn't associative. For example, if a, b, and c are **float**s, then $(a+b)+c$ may not be exactly equal to $a+(b+c)$. See Exercise 5.5.

When a variable is included in a `reduction` clause, the variable itself is shared. However, a private variable is created for each thread in the team. In the `parallel` block each time a thread executes a statement involving the variable, it uses the private variable. When the `parallel` block ends, the values in the private variables are combined into the shared variable. Thus our latest version of the code

```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count) \
      reduction(+: global_result)
   global_result += Local_trap(double a, double b, int n);
```

effectively executes code that is identical to our previous version:

```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count)
   {
      double my_result = 0.0;  /* private */
      my_result += Local_trap(double a, double b, int n);
#     pragma omp critical
      global_result += my_result;
   }
```

One final point to note is that the threads' private variables are initialized to 0. This is analogous to our initializing `my_result` to zero. In general, the private variables created for a `reduction` clause are initialized to the *identity value* for the operator. For example, if the operator is multiplication, the private variables would be initialized to 1. See Table 5.1 for the entire list.

Table 5.1 Identity values for the various reduction operators in OpenMP.

Operator	Identity Value
+	0
*	1
-	0
&	~0
|	0
^	0
&&	1
||	0

5.5 The parallel for directive

As an alternative to our explicit parallelization of the trapezoidal rule, OpenMP provides the parallel **for** directive. Using it, we can parallelize the serial trapezoidal rule

```
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++)
   approx += f(a + i*h);
approx = h*approx;
```

by simply placing a directive immediately before the **for** loop:

```
   h = (b-a)/n;
   approx = (f(a) + f(b))/2.0;
#  pragma omp parallel for num_threads(thread_count) \
      reduction(+: approx)
   for (i = 1; i <= n-1; i++)
      approx += f(a + i*h);
   approx = h*approx;
```

Like the parallel directive, the parallel **for** directive forks a team of threads to execute the following structured block. However, the structured block following the parallel **for** directive must be a **for** loop. Furthermore, with the parallel **for** directive the system parallelizes the **for** loop by dividing the iterations of the loop among the threads. So the parallel **for** directive is therefore very different from the parallel directive, because in a block that is preceded by a parallel directive, in general, the work must be divided among the threads by the threads themselves.

In a **for** loop that has been parallelized with a parallel **for** directive, the default partitioning of the iterations among the threads is up to the system. However, most systems use roughly a block partitioning, that is, if there are m iterations, then roughly the first m/thread_count are assigned to thread 0, the next m/thread_count are assigned to thread 1, and so on.

5.5.3 Finding loop-carried dependences

Perhaps the first thing to observe is that when we're attempting to use a `parallel` **for** directive, we only need to worry about loop-carried dependences. We don't need to worry about more general data dependences. For example, in the loop

```
1     for (i = 0; i < n; i++) {
2        x[i] = a + i*h;
3        y[i] = exp(x[i]);
4     }
```

there is a data dependence between Lines 2 and 3. However, there is no problem with the parallelization

```
1  #  pragma omp parallel for num_threads(thread_count)
2     for (i = 0; i < n; i++) {
3        x[i] = a + i*h;
4        y[i] = exp(x[i]);
5     }
```

since the computation of `x[i]` and its subsequent use will always be assigned to the same thread.

Also observe that at least one of the statements must write or update the variable in order for the statements to represent a dependence, so to detect a loop-carried dependence, we should only concern ourselves with variables that are updated by the loop body. That is, we should look for variables that are read or written in one iteration, and written in another. Let's look at a couple of examples.

5.5.4 Estimating π

One way to get a numerical approximation to π is to use many terms in the formula[4]

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}.$$

We can implement this formula in serial code with

```
1        double factor = 1.0;
2        double sum = 0.0;
3        for (k = 0; k < n; k++) {
4           sum += factor/(2*k+1);
5           factor = -factor;
6        }
7        pi_approx = 4.0*sum;
```

[4] This is by no means the best method for approximating π, since it requires a *lot* of terms to get a reasonably accurate result. However, in this case, lots of terms will be better to demonstrate the effects of parallelism, and we're more interested in the formula itself than the actual estimate.

Table 5.3 Odd-even sort with two `parallel` **for** directives and two **for** directives. Times are in seconds.

thread_count	1	2	3	4
Two `parallel` **for** directives	0.770	0.453	0.358	0.305
Two **for** directives	0.732	0.376	0.294	0.239

5.7 Scheduling loops

When we first encountered the `parallel` **for** directive, we saw that the exact assignment of loop iterations to threads is system dependent. However most OpenMP implementations use roughly a block partitioning: if there are n iterations in the serial loop, then in the parallel loop the first $n/$`thread_count` are assigned to thread 0, the next $n/$`thread_count` are assigned to thread 1, and so on. It's not difficult to think of situations in which this assignment of iterations to threads would be less than optimal. For example, suppose we want to parallelize the loop

```
sum = 0.0;
for (i = 0; i <= n; i++)
   sum += f(i);
```

Also suppose that the time required by the call to f is proportional to the size of the argument `i`. Then a block partitioning of the iterations will assign much more work to thread `thread_count` -1 than it will assign to thread 0. A better assignment of work to threads might be obtained with a **cyclic** partitioning of the iterations among the threads. In a cyclic partitioning, the iterations are assigned, one at a time, in a "round-robin" fashion to the threads. Suppose $t =$ `thread_count`. Then a cyclic partitioning will assign the iterations as follows:

Thread	Iterations
0	$0,\ n/t,\ 2n/t,\ \ldots$
1	$1,\ n/t+1,\ 2n/t+1,\ \ldots$
$\vdots$	$\vdots$
$t-1$	$t-1,\ n/t+t-1,\ 2n/t+t-1,\ \ldots$

To get a feel for how drastically this can affect performance, we wrote a program in which we defined

```
double f(int i) {
   int j, start = i*(i+1)/2, finish = start + i;
   double return_val = 0.0;

   for (j = start; j <= finish; j++) {
      return_val += sin(j);
   }
   return return_val;
}  /* f */
```

The call $f(i)$ calls the sin function i times, and, for example, the time to execute $f(2i)$ requires approximately twice as much time as the time to execute $f(i)$.

When we ran the program with $n = 10{,}000$ and one thread, the run-time was 3.67 seconds. When we ran the program with two threads and the default assignment—iterations 0–5000 on thread 0 and iterations 5001–10,000 on thread 1—the run-time was 2.76 seconds. This is a speedup of only 1.33. However, when we ran the program with two threads and a cyclic assignment, the run-time was decreased to 1.84 seconds. This is a speedup of 1.99 over the one-thread run and a speedup of 1.5 over the two-thread block partition!

We can see that a good assignment of iterations to threads can have a very significant effect on performance. In OpenMP, assigning iterations to threads is called **scheduling**, and the `schedule` clause can be used to assign iterations in either a `parallel` **for** or a **for** directive.

5.7.1 The `schedule` clause

In our example, we already know how to obtain the default schedule: we just add a `parallel` **for** directive with a `reduction` clause:

```
      sum = 0.0;
#     pragma omp parallel for num_threads(thread_count) \
         reduction(+:sum)
      for (i = 0; i <= n; i++)
         sum += f(i);
```

To get a cyclic schedule, we can add a `schedule` clause to the `parallel` **for** directive:

```
      sum = 0.0;
#     pragma omp parallel for num_threads(thread_count) \
         reduction(+:sum) schedule(static,1)
      for (i = 0; i <= n; i++)
         sum += f(i);
```

In general, the `schedule` clause has the form

```
schedule(<type> [, <chunksize>])
```

The `type` can be any one of the following:

- `static`. The iterations can be assigned to the threads before the loop is executed.
- `dynamic` or `guided`. The iterations are assigned to the threads while the loop is executing, so after a thread completes its current set of iterations, it can request more from the run-time system.
- `auto`. The compiler and/or the run-time system determine the schedule.
- `runtime`. The schedule is determined at run-time based on an environment variable (more on this later).

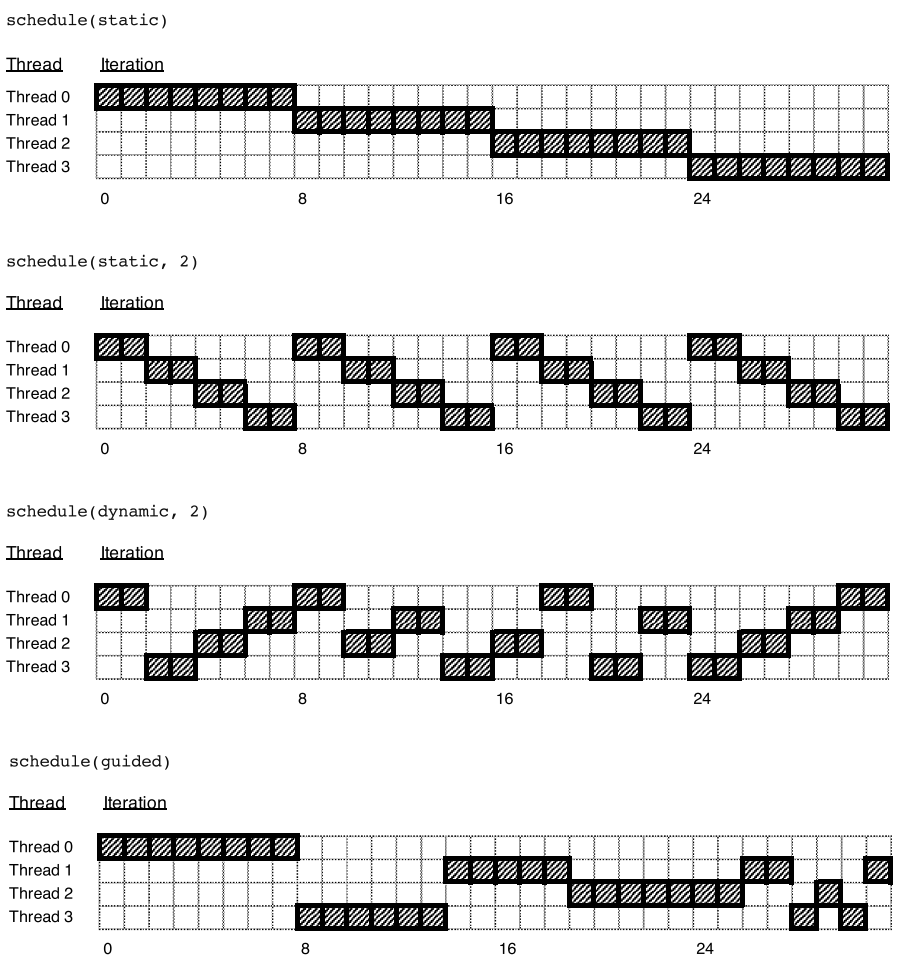

FIGURE 5.5

Scheduling visualization for the `static`, `dynamic`, and `guided` schedule types with 4 threads and 32 iterations. The first static schedule uses the default `chunksize`, whereas the second uses a `chunksize` of **2**. The exact distribution of work across threads will vary between different executions of the program for the `dynamic` and `guided` schedule types, so this visualization shows one of many possible scheduling outcomes.

The `chunksize` is a positive integer. In OpenMP parlance, a **chunk** of iterations is a block of iterations that would be executed consecutively in the serial loop. The number of iterations in the block is the `chunksize`. Only `static`, `dynamic`, and `guided` schedules can have a `chunksize`. This determines the details of the schedule, but its exact interpretation depends on the `type`. Fig. 5.5 provides a visualization of how work is scheduled using the static, dynamic, and guided types.

5.7.2 The `static` schedule type

For a `static` schedule, the system assigns chunks of `chunksize` iterations to each thread in a round-robin fashion. As an example, suppose we have 12 iterations, 0, 1, . . . , 11, and three threads. Then if `schedule(`**static** , 1) is used, in the `parallel` **for** or **for** directive, we've already seen that the iterations will be assigned as

Thread 0 : 0, 3, 6, 9
Thread 1 : 1, 4, 7, 10
Thread 2 : 2, 5, 8, 11

If `schedule(`**static** , 2) is used, then the iterations will be assigned as

Thread 0 : 0, 1, 6, 7
Thread 1 : 2, 3, 8, 9
Thread 2 : 4, 5, 10, 11

If `schedule(`**static** , 4) is used, the iterations will be assigned as

Thread 0 : 0, 1, 2, 3
Thread 1 : 4, 5, 6, 7
Thread 2 : 8, 9, 10, 11

The default schedule is defined by your particular implementation of OpenMP, but in most cases it is equivalent to the clause

```
schedule(static, total_iterations / thread_count)
```

It is also worth noting that the `chunksize` can be omitted. If omitted, the `chunksize` is approximately `total_iterations / thread_count`.

The `static` schedule is a good choice when each loop iteration takes roughly the same amount of time to compute. It also has the advantage that threads in subsequent loops with the same number of iterations will be assigned to the same ranges; this can improve the speed of memory accesses, particularly on NUMA systems (see Chapter 2).

5.7.3 The `dynamic` and `guided` schedule types

In a `dynamic` schedule, the iterations are also broken up into chunks of `chunksize` consecutive iterations. Each thread executes a chunk, and when a thread finishes a chunk, it requests another one from the run-time system. This continues until all the iterations are completed. The `chunksize` can be omitted. When it is omitted, a `chunksize` of 1 is used.

The primary difference between `static` and `dynamic` schedules is that the `dynamic` schedule assigns ranges to threads on a first-come, first-served basis. This can be advantageous if loop iterations do not take a uniform amount of time to compute (some

Table 5.4 Assignment of trapezoidal rule iterations 1–9999 using a guided schedule with two threads.

Thread	Chunk	Size of Chunk	Remaining Iterations
0	1 – 5000	5000	4999
1	5001 – 7500	2500	2499
1	7501 – 8750	1250	1249
1	8751 – 9375	625	624
0	9376 – 9687	312	312
1	9688 – 9843	156	156
0	9844 – 9921	78	78
1	9922 – 9960	39	39
1	9961 – 9980	20	19
1	9981 – 9990	10	9
1	9991 – 9995	5	4
0	9996 – 9997	2	2
1	9998 – 9998	1	1
0	9999 – 9999	1	0

algorithms are more compute-intensive in later iterations, for instance). However, since the ranges are not allocated ahead of time, there is some overhead associated with assigning them dynamically at run-time. Increasing the chunk size strikes a balance between the performance characteristics of `static` and `dynamic` scheduling; with larger chunk sizes, fewer dynamic assignments will be made.

The `guided` schedule is similar to `dynamic` in that each thread also executes a chunk and requests another one when it's finished. However, in a `guided` schedule, as chunks are completed, the size of the new chunks decreases. For example, on one of our systems, if we run the trapezoidal rule program with the `parallel` **for** directive and a `schedule(guided)` clause, then when $n = 10{,}000$ and `thread_count` $= 2$, the iterations are assigned as shown in Table 5.4. We see that the size of the chunk is approximately the number of iterations remaining divided by the number of threads. The first chunk has size $9999/2 \approx 5000$, since there are 9999 unassigned iterations. The second chunk has size $4999/2 \approx 2500$, and so on.

In a `guided` schedule, if no `chunksize` is specified, the size of the chunks decreases down to 1. If `chunksize` is specified, it decreases down to `chunksize`, with the exception that the very last chunk can be smaller than `chunksize`. The `guided` schedule can improve the balance of load across threads when later iterations are more compute-intensive.

5.7.4 The `runtime` schedule type

To understand `schedule(runtime)`, we need to digress for a moment and talk about **environment variables**. As the name suggests, environment variables are named values that can be accessed by a running program. That is, they're available in the program's

environment. Some commonly used environment variables are `PATH`, `HOME`, and `SHELL`. The `PATH` variable specifies which directories the shell should search when it's looking for an executable and is usually defined in both Unix and Windows. The `HOME` variable specifies the location of the user's home directory, and the `SHELL` variable specifies the location of the executable for the user's shell. These are usually defined in Unix systems. In both Unix-like systems (e.g., Linux and macOS) and Windows, environment variables can be examined and specified on the command line. In Unix-like systems, you can use the shell's command line. In Windows systems, you can use the command line in an integrated development environment.

As an example, if we're using the bash shell (one of the most common Unix shells), we can examine the value of an environment variable by typing:

```
$ echo $PATH
```

and we can use the `export` command to set the value of an environment variable:

```
$ export TEST_VAR="hello"
```

These commands also work on `ksh`, `sh`, and `zsh`. For details about how to examine and set environment variables for your particular system, check the `man` pages for your shell, or consult with your system administrator or local expert.

When `schedule(runtime)` is specified, the system uses the environment variable `OMP_SCHEDULE` to determine at run-time how to schedule the loop. The `OMP_SCHEDULE` environment variable can take on any of the values that can be used for a static, dynamic, or guided schedule. For example, suppose we have a `parallel` **for** directive in a program and it has been modified by `schedule(runtime)`. Then if we use the bash shell, we can get a cyclic assignment of iterations to threads by executing the command

```
$ export OMP_SCHEDULE="static,1"
```

Now, when we start executing our program, the system will schedule the iterations of the **for** loop as if we had the clause `schedule(static,1)` modifying the `parallel` **for** directive. This can be very useful for testing a variety of scheduling configurations.

The following bash shell script demonstrates how one might take advantage of this environment variable to test a range of schedules and chunk sizes. It runs a matrix-vector multiplication program that has a `parallel` **for** directive with the `schedule(runtime)` clause.

```
#!/usr/bin/env bash

declare -a schedules=("static" "dynamic" "guided")
declare -a chunk_sizes=("" 1000 100 10 1)

for schedule in "${schedules[@]}"; do
   echo "Schedule: ${schedule}"
   for chunk_size in "${chunk_sizes[@]}"; do
      echo "  Chunk Size: ${chunk_size}"
      sched_param="${schedule}"
```

```
        if [[ "${chunk_size}" != "" ]]; then
            # A blank string indicates we want
            # the default chunk size
            sched_param="${schedule},${chunk_size}"
        fi

        # Run the program with OMP_SCHEDULE set:
        OMP_SCHEDULE="${sched_param}" ./omp_mat_vect 4 2500 2500
    done
    echo
done
```

5.7.5 Which schedule?

If we have a **for** loop that we're able to parallelize, how do we decide which type of schedule we should use and what the `chunksize` should be? As you may have guessed, there *is* some overhead associated with the use of a `schedule` clause. Furthermore, the overhead is greater for `dynamic` schedules than `static` schedules, and the overhead associated with `guided` schedules is the greatest of the three. Thus if we're getting satisfactory performance without a `schedule` clause, we should go no further. However, if we suspect that the performance of the default schedule can be substantially improved, we should probably experiment with some different schedules.

In the example at the beginning of this section, when we switched from the default schedule to `schedule(`**static** `,1)`, the speedup of the two-threaded execution of the program increased from 1.33 to 1.99. Since it's *extremely* unlikely that we'll get speedups that are significantly better than 1.99, we can just stop here, at least if we're only going to use two threads with 10,000 iterations. If we're going to be using varying numbers of threads and varying numbers of iterations, we need to do more experimentation, and it's entirely possible that we'll find that the optimal schedule depends on both the number of threads and the number of iterations.

It can also happen that we'll decide that the performance of the default schedule isn't very good, and we'll proceed to search through a large array of schedules and iteration counts only to conclude that our loop doesn't parallelize very well and *no* schedule is going to give us much improved performance. For an example of this, see Programming Assignment 5.4.

There are some situations in which it's a good idea to explore some schedules before others:

- If each iteration of the loop requires roughly the same amount of computation, then it's likely that the default distribution will give the best performance.
- If the cost of the iterations decreases (or increases) linearly as the loop executes, then a `static` schedule with small chunksizes will probably give the best performance.
- If the cost of each iteration can't be determined in advance, then it may make sense to explore a variety of scheduling options. The `schedule(runtime)` clause can

be used here, and the different options can be explored by running the program with different assignments to the environment variable `OMP_SCHEDULE`.

5.8 Producers and consumers

Let's take a look at a parallel problem that isn't amenable to parallelization using a `parallel` **for** or **for** directive.

5.8.1 Queues

Recall that a **queue** is a list abstract datatype in which new elements are inserted at the "rear" of the queue and elements are removed from the "front" of the queue. A queue can thus be viewed as an abstraction of a line of customers waiting to pay for their groceries in a supermarket. The elements of the list are the customers. New customers go to the end or "rear" of the line, and the next customer to check out is the customer standing at the "front" of the line.

When a new entry is added to the rear of a queue, we sometimes say that the entry has been "enqueued," and when an entry is removed from the front of a queue, we sometimes say that the entry has been "dequeued."

Queues occur frequently in computer science. For example, if we have a number of processes, each of which wants to store some data on a hard drive, then a natural way to ensure that only one process writes to the disk at a time is to have the processes form a queue, that is, the first process that wants to write gets access to the drive first, the second process gets access to the drive next, and so on.

A queue is also a natural data structure to use in many multithreaded applications. For example, suppose we have several "producer" threads and several "consumer" threads. The producer threads might "produce" requests for data from a server—for example, current stock prices—while the consumer threads might "consume" the request by finding or generating the requested data—the current stock prices. The producer threads could enqueue the requested prices, and the consumer threads could dequeue them. In this example, the process wouldn't be completed until the consumer threads had given the requested data to the producer threads.

5.8.2 Message-passing

Another natural application would be implementing message-passing on a shared-memory system. Each thread could have a shared-message queue, and when one thread wanted to "send a message" to another thread, it could enqueue the message in the destination thread's queue. A thread could receive a message by dequeuing the message at the head of its message queue.

Let's implement a relatively simple message-passing program, in which each thread generates random integer "messages" and random destinations for the messages. After creating the message, the thread enqueues the message in the appropriate

message queue. After sending a message, a thread checks its queue to see if it has received a message. If it has, it dequeues the first message in its queue and prints it out. Each thread alternates between sending and trying to receive messages. We'll let the user specify the number of messages each thread should send. When a thread is done sending messages, it receives messages until all the threads are done, at which point all the threads quit. Pseudocode for each thread might look something like this:

```
for (sent_msgs = 0; sent_msgs < send_max; sent_msgs++) {
   Send_msg();
   Try_receive();
}

while (!Done())
   Try_receive();
```

5.8.3 Sending messages

Note that accessing a message queue to enqueue a message is probably a critical section. Although we haven't looked into the details of the implementation of the message queue, it seems likely that we'll want to have a variable that keeps track of the rear of the queue. For example, if we use a singly linked list with the tail of the list corresponding to the rear of the queue, then, to efficiently enqueue, we would want to store a pointer to the rear. When we enqueue a new message, we'll need to check and update the rear pointer. If two threads try to do this simultaneously, we may lose a message that has been enqueued by one of the threads. (It might help to draw a picture!) The results of the two operations will conflict, and hence enqueuing a message will form a critical section.

Pseudocode for the `Send_msg()` function might look something like this:

```
   mesg = random();
   dest = random() % thread_count;
#  pragma omp critical
   Enqueue(queue, dest, my_rank, mesg);
```

Note that this allows a thread to send a message to itself.

5.8.4 Receiving messages

The synchronization issues for receiving a message are a little different. Only the owner of the queue (that is, the destination thread) will dequeue from a given message queue. As long as we dequeue one message at a time, if there are at least two messages in the queue, a call to `Dequeue` can't possibly conflict with any calls to `Enqueue`. So if we keep track of the size of the queue, we can avoid any synchronization (for example, `critical` directives), as long as there are at least two messages.

Now you may be thinking, "What about the variable storing the size of the queue?" This would be a problem if we simply store the size of the queue. How-

ever, if we store two variables, `enqueued` and `dequeued`, then the number of messages in the queue is

```
queue_size = enqueued - dequeued
```

and the only thread that will update `dequeued` is the owner of the queue. Observe that one thread can update `enqueued` at the same time that another thread is using it to compute `queue_size`. To see this, let's suppose thread q is computing `queue_size`. It will either get the old value of `enqueued` or the new value. It *may* therefore compute a `queue_size` of 0 or 1 when `queue_size` should actually be 1 or 2, respectively, but in our program this will only cause a modest delay. Thread q will try again later if `queue_size` is 0 when it should be 1, and it will execute the critical section directive unnecessarily if `queue_size` is 1 when it should be 2.

Thus we can implement `Try_receive` as follows:

```
   queue_size = enqueued - dequeued;
   if (queue_size == 0) return;
   else if (queue_size == 1)
#     pragma omp critical
      Dequeue(queue, &src, &mesg);
   else
      Dequeue(queue, &src, &mesg);
   Print_message(src, mesg);
```

5.8.5 Termination detection

We also need to think about implementation of the `Done` function. First note that the following "obvious" implementation will have problems:

```
queue_size = enqueued - dequeued;
if (queue_size == 0)
   return TRUE;
else
   return FALSE;
```

If thread u executes this code, it's entirely possible that some thread—call it thread v—will send a message to thread u *after* u has computed `queue_size` = 0. Of course, after thread u computes `queue_size` = 0, it will terminate and the message sent by thread v will never be received.

However, in our program, after each thread has completed the **for** loop, it won't send any new messages. Thus if we add a counter `done_sending`, and each thread increments this after completing its **for** loop, then we *can* implement `Done` as follows:

```
queue_size = enqueued - dequeued;
if (queue_size == 0 && done_sending == thread_count)
   return TRUE;
else
   return FALSE;
```

5.8.6 Startup

When the program begins execution, a single thread, the master thread, will get command-line arguments and allocate an array of message queues, one for each thread. This array needs to be shared among the threads, since any thread can send to any other thread, and hence any thread can enqueue a message in any of the queues. Given that a message queue will (at a minimum) store

- a list of messages,
- a pointer or index to the rear of the queue,
- a pointer or index to the front of the queue,
- a count of messages enqueued, and
- a count of messages dequeued,

it makes sense to store the queue in a struct, and to reduce the amount of copying when passing arguments, it also makes sense to make the message queue an array of pointers to structs. Thus once the array of queues is allocated by the master thread, we can start the threads using a `parallel` directive, and each thread can allocate storage for its individual queue.

An important point here is that one or more threads may finish allocating their queues before some other threads. If this happens, the threads that finish first could start trying to enqueue messages in a queue that hasn't been allocated and cause the program to crash. We therefore need to make sure that none of the threads starts sending messages until all the queues are allocated. Recall that we've seen that several OpenMP directives provide implicit barriers when they're completed, that is, no thread will proceed past the end of the block until all the threads in the team have completed the block. In this case, though, we'll be in the middle of a `parallel` block, so we can't rely on an implicit barrier from some other OpenMP construct—we need an *explicit* barrier. Fortunately, OpenMP provides one:

```
# pragma omp barrier
```

When a thread encounters the barrier, it blocks until all the threads in the team have reached the barrier. After all the threads have reached the barrier, all the threads in the team can proceed.

5.8.7 The `atomic` directive

After completing its sends, each thread increments `done_sending` before proceeding to its final loop of receives. Clearly, incrementing `done_sending` is a critical section, and we could protect it with a `critical` directive. However, OpenMP provides a potentially higher performance directive: the `atomic` directive[5]:

```
# pragma omp atomic
```

[5] OpenMP provides several clauses that modify the behavior of the `atomic` directive. We're describing the default `atomic` directive, which is the same as an `atomic` directive with an `update` clause. See [47].

Unlike the `critical` directive, it can only protect critical sections that consist of a single C assignment statement. Further, the statement must have one of the following forms:

```
x <op>= <expression>;
x++;
++x;
x--;
--x;
```

Here <op> can be one of the binary operators

```
+, *, -, /, &, ^, |, <<, or >>.
```

It's also important to remember that <expression> must not reference `x`.

It should be noted that only the load and store of `x` are guaranteed to be protected. For example, in the code

```
#       pragma omp atomic
        x += y++;
```

a thread's update to `x` will be completed before any other thread can begin updating `x`. However, the update to `y` may be unprotected and the results may be unpredictable.

The idea behind the `atomic` directive is that many processors provide a special load-modify-store instruction, and a critical section that only does a load-modify-store can be protected much more efficiently by using this special instruction rather than the constructs that are used to protect more general critical sections.

5.8.8 Critical sections and locks

To finish our discussion of the message-passing program, we need to take a more careful look at OpenMP's specification of the `critical` directive. In our earlier examples, our programs had at most one critical section, and the `critical` directive forced mutually exclusive access to the section by all the threads. In this program, however, the use of critical sections is more complex. If we simply look at the source code, we'll see three blocks of code preceded by a `critical` or an `atomic` directive:

- `done_sending++;`
- `Enqueue(q_p, my_rank, mesg);`
- `Dequeue(q_p, &src, &mesg);`

However, we don't need to enforce exclusive access across all three of these blocks of code. We don't even need to enforce completely exclusive access within `Enqueue` and `Dequeue`. For example, it would be fine for, say, thread 0 to enqueue a message in thread 1's queue at the same time that thread 1 is enqueuing a message in thread 2's queue. But for the second and third blocks—the blocks protected by `critical` directives—this is exactly what OpenMP does. From OpenMP's point of view our program has two distinct critical sections: the critical section protected by the `atomic`

directive, (`done_sending++`), and the "composite" critical section in which we enqueue and dequeue messages.

Since enforcing mutual exclusion among threads serializes execution, this default behavior of OpenMP—treating all critical blocks as part of one composite critical section—can be highly detrimental to our program's performance. OpenMP *does* provide the option of adding a name to a critical directive:

```
# pragma omp critical(name)
```

When we do this, two blocks protected with `critical` directives with different names *can* be executed simultaneously. However, the names are set during compilation, and we want a different critical section for each thread's queue. Therefore we need to set the names at run-time, and in our setting, when we want to allow simultaneous access to the same block of code by threads accessing different queues, the named `critical` directive isn't sufficient.

The alternative is to use **locks**.[6] A lock consists of a data structure and functions that allow the programmer to explicitly enforce mutual exclusion in a critical section. The use of a lock can be roughly described by the following pseudocode:

```
/* Executed by one thread */
Initialize the lock data structure;
. . .
/* Executed by multiple threads */
Attempt to lock or set the lock data structure;
Critical section;
Unlock or unset the lock data structure;
. . .
/* Executed by one thread */
Destroy the lock data structure;
```

The lock data structure is shared among the threads that will execute the critical section. One of the threads (e.g., the master thread) will initialize the lock, and when all the threads are done using the lock, one of the threads should destroy it.

Before a thread enters the critical section, it attempts to *set* the lock by calling the lock function. If no other thread is executing code in the critical section, it *acquires* the lock and proceeds into the critical section past the call to the lock function. When the thread finishes the code in the critical section, it calls an unlock function, which *releases* or *unsets* the lock and allows another thread to acquire the lock.

While a thread owns the lock, no other thread can enter the critical section. If another thread attempts to enter the critical section, it will *block* when it calls the lock function. If multiple threads are blocked in a call to the lock function, then when the thread in the critical section releases the lock, one of the blocked threads returns from the call to the lock, and the others remain blocked.

[6] If you've studied the Pthreads chapter, you've already learned about locks, and you can skip ahead to the syntax for OpenMP locks.

OpenMP has two types of locks: **simple** locks and **nested** locks. A simple lock can only be set once before it is unset, while a nested lock can be set multiple times by the same thread before it is unset. The type of an OpenMP simple lock is `omp_lock_t`, and the simple lock functions that we'll be using are

```
void omp_init_lock(omp_lock_t*     lock_p   /* out */);
void omp_set_lock(omp_lock_t*      lock_p   /* in/out */);
void omp_unset_lock(omp_lock_t*    lock_p   /* in/out */);
void omp_destroy_lock(omp_lock_t* lock_p    /* in/out */);
```

The type and the functions are specified in `omp.h`. The first function initializes the lock so that it's unlocked, that is, no thread owns the lock. The second function attempts to set the lock. If it succeeds, the calling thread proceeds; if it fails, the calling thread blocks until the lock becomes available. The third function unsets the lock so another thread can acquire it. The fourth function makes the lock uninitialized. We'll only use simple locks. For information about nested locks, see [9], [10], or [47].

5.8.9 Using locks in the message-passing program

In our earlier discussion of the limitations of the `critical` directive, we saw that in the message-passing program, we wanted to ensure mutual exclusion in each individual message queue, not in a particular block of source code. Locks allow us to do this. If we include a data member with type `omp_lock_t` in our queue struct, we can simply call `omp_set_lock` each time we want to ensure exclusive access to a message queue. So the code

```
#  pragma omp critical
   /* q_p = msg_queues[dest] */
   Enqueue(q_p, my_rank, mesg);
```

can be replaced with

```
/* q_p = msg_queues[dest] */
omp_set_lock(&q_p->lock);
Enqueue(q_p, my_rank, mesg);
omp_unset_lock(&q_p->lock);
```

Similarly, the code

```
#  pragma omp critical
   /* q_p = msg_queues[my_rank] */
   Dequeue(q_p, &src, &mesg);
```

can be replaced with

```
/* q_p = msg_queues[my_rank] */
omp_set_lock(&q_p->lock);
Dequeue(q_p, &src, &mesg);
omp_unset_lock(&q_p->lock);
```

Now when a thread tries to send or receive a message, it can only be blocked by a thread attempting to access the same message queue, since different message queues have different locks. In our original implementation, only one thread could send at a time, regardless of the destination.

Note that it would also be possible to put the calls to the lock functions in the queue functions `Enqueue` and `Dequeue`. However, to preserve the performance of `Dequeue`, we would also need to move the code that determines the size of the queue (`enqueued – dequeued`) to `Dequeue`. Without it, the `Dequeue` function will lock the queue every time it is called by `Try_receive`. In the interest of preserving the structure of the code we've already written, we'll leave the calls to `omp_set_lock` and `omp_unset_lock` in the `Send` and `Try_receive` functions.

Since we're now including the lock associated with a queue in the queue struct, we can add initialization of the lock to the function that initializes an empty queue. Destruction of the lock can be done by the thread that owns the queue before it frees the queue.

5.8.10 `Critical` directives, `atomic` directives, or locks?

Now that we have three mechanisms for enforcing mutual exclusion in a critical section, it's natural to wonder when one method is preferable to another. In general, the `atomic` directive has the potential to be the fastest method of obtaining mutual exclusion. Thus if your critical section consists of an assignment statement having the required form, it will probably perform at least as well with the `atomic` directive as the other methods. However, the OpenMP specification [47] allows the `atomic` directive to enforce mutual exclusion across *all* `atomic` directives in the program—this is the way the unnamed `critical` directive behaves. If this might be a problem—for example, you have multiple different critical sections protected by `atomic` directives—you should use named `critical` directives or locks. For example, suppose we have a program in which it's possible that one thread will execute the code on the left while another executes the code on the right.

```
# pragma omp atomic
  x++;
```

```
# pragma omp atomic
  y++;
```

Even if `x` and `y` are unrelated memory locations, it's possible that if one thread is executing `x++`, then no thread can simultaneously execute `y++`. It's important to note that the standard doesn't require this behavior. If two statements are protected by `atomic` directives and the two statements modify different variables, then there are implementations that treat the two statements as different critical sections. (See Exercise 5.10.) On the other hand, different statements that modify the same variable *will* be treated as if they belong to the same critical section, regardless of the implementation.

We've already seen some limitations to the use of `critical` directives. However, both named and unnamed `critical` directives are very easy to use. Furthermore, in the implementations of OpenMP that we've used there doesn't seem to be a very large difference between the performance of critical sections protected by a critical directive, and `critical` sections protected by locks, so if you can't use an `atomic`

directive, but you can use a `critical` directive, you probably should. Thus the use of locks should probably be reserved for situations in which mutual exclusion is needed for a data structure rather than a block of code.

5.8.11 Some caveats

You should exercise caution when using the mutual exclusion techniques we've discussed. They can definitely cause serious programming problems. Here are a few things to be aware of:

1. You shouldn't mix the different types of mutual exclusion for a single critical section. For example, suppose a program contains the following two segments:

```
#  pragma omp atomic
   x += f(y);

#  pragma omp critical
   x = g(x);
```

   The update to `x` on the right doesn't have the form required by the `atomic` directive, so the programmer used a `critical` directive. However, the `critical` directive won't exclude the action executed by the `atomic` block, and it's possible that the results will be incorrect. The programmer needs to either rewrite the function *g* so that its use can have the form required by the `atomic` directive or to protect both blocks with a `critical` directive.
2. There is no guarantee of **fairness** in mutual exclusion constructs. This means that it's possible that a thread can be blocked forever in waiting for access to a critical section. For example, in the code

```
   while(1) {
      . . .
#     pragma omp critical
      x = g(my_rank);
      . . .
   }
```

   it's possible that, for example, thread 1 can block forever waiting to execute `x = g(my_rank)` while the other threads repeatedly execute the assignment. Of course, this wouldn't be an issue if the loop terminated.
3. It can be dangerous to "nest" mutual exclusion constructs. As an example, suppose a program contains the following two segments:

```
#  pragma omp critical
   y = f(x);
   . . .
   double f(double x) {
#     pragma omp critical
      z = g(x);  /* z is shared */
      . . .
   }
```

This is guaranteed to **deadlock**. When a thread attempts to enter the second critical section, it will block forever. If thread *u* is executing code in the first critical block, no thread can execute code in the second block. In particular, thread *u* can't execute this code. However, if thread *u* is blocked waiting to enter the second critical block, then it will never leave the first, and it will stay blocked forever.
In this example, we can solve the problem by using named critical sections. That is, we could rewrite the code as

```
#   pragma omp critical(one)
    y = f(x);
    . . .
    double f(double x) {
#       pragma omp critical(two)
        z = g(x);  /* z is global */
        . . .
    }
```

However, it's not difficult to come up with examples when naming won't help. For example, if a program has two named critical sections—say `one` and `two`—and threads can attempt to enter the critical sections in different orders, then deadlock can occur. For example, suppose thread *u* enters `one` at the same time that thread *v* enters `two` and *u* then attempts to enter `two` while *v* attempts to enter `one`:

Time	Thread *u*	Thread *v*
0	Enter crit. sect. `one`	Enter crit. sect. `two`
1	Attempt to enter `two`	Attempt to enter `one`
2	Block	Block

Then both *u* and *v* will block forever waiting to enter the critical sections. So it's not enough to just use different names for the critical sections—the programmer must ensure that different critical sections are always entered in the same order.

5.9 Caches, cache coherence, and false sharing[7]

Recall that for a number of years now, computers have been able to execute operations involving only the processor much faster than they can access data in main memory. If a processor must read data from main memory for each operation, it will spend most of its time simply waiting for the data from memory to arrive. Also recall that to address this problem, chip designers have added blocks of relatively fast memory to processors. This faster memory is called **cache memory**.

[7] This material is also covered in Chapter 4. So if you've already read that chapter, you may want to just skim this section.

Table 5.5 Memory and cache accesses.

Time	Memory	Th 0	Th 0 cache	Th 1	Th 1 cache
0	x = 5	Load x	—	Load x	—
1	x = 5	—	x = 5	—	x = 5
2	x = 5	x++	x = 5	—	x = 5
3	???	—	x = 6	my_z = x	???

The design of cache memory takes into consideration the principles of **temporal and spatial locality**: if a processor accesses main memory location x at time t, then it is likely that at times close to t it will access main memory locations close to x. Thus if a processor needs to access main memory location x, rather than transferring only the contents of x to/from main memory, a block of memory containing x is transferred from/to the processor's cache. Such a block of memory is called a **cache line** or **cache block**.

In Section 2.3.5, we saw that the use of cache memory can have a huge impact on shared memory. Let's recall why. First, consider the following situation: Suppose x is a shared variable with the value five, and both thread 0 and thread 1 read x from memory into their (separate) caches, because both want to execute the statement

```
my_y = x;
```

Here, my_y is a private variable defined by both threads. Now suppose thread 0 executes the statement

```
x++;
```

Finally, suppose that thread 1 now executes

```
my_z = x;
```

where my_z is another private variable. Table 5.5 illustrates the sequence of accesses.

What's the value in my_z? Is it five? Or is it six? The problem is that there are (at least) three copies of x: the one in main memory, the one in thread 0's cache, and the one in thread 1's cache. When thread 0 executed x++, what happened to the values in main memory and thread 1's cache? This is the **cache coherence** problem, which we discussed in Chapter 2. We saw there that most systems insist that the caches be made aware that changes have been made to data they are caching. The line in the cache of thread 1 would have been marked *invalid* when thread 0 executed x++, and before assigning my_z = x, the core running thread 1 would see that its value of x was out of date. Thus the core running thread 0 would have to update the copy of x in main memory (either now or earlier), and the core running thread 1 could get the line with the updated value of x from main memory. For further details, see Chapter 2.

The use of cache coherence can have a dramatic effect on the performance of shared-memory systems. To illustrate this, let's take a look at matrix-vector multiplication. Recall that if $A = (a_{ij})$ is an $m \times n$ matrix and $\mathbf{x}$ is a vector with n components, then their product $\mathbf{y} = A\mathbf{x}$ is a vector with m components, and its ith component y_i is

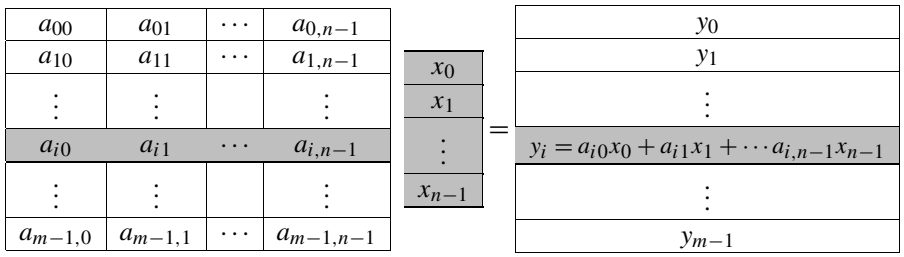

FIGURE 5.6

Matrix-vector multiplication.

found by forming the dot product of the ith row of A with $\mathbf{x}$:

$$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}.$$

See Fig. 5.6.

So if we store A as a two-dimensional array and $\mathbf{x}$ and $\mathbf{y}$ as one-dimensional arrays, we can implement serial matrix-vector multiplication with the following code:

```
for (i = 0; i < m; i++) {
   y[i] = 0.0;
   for (j = 0; j < n; j++)
      y[i] += A[i][j]*x[j];
}
```

There are no loop-carried dependences in the outer loop, since `A` and `x` are never updated and iteration i only updates `y[i]`. Thus we can parallelize this by dividing the iterations in the outer loop among the threads:

```
1  #  pragma omp parallel for num_threads(thread_count)  \
2        default(none) private(i, j) shared(A, x, y, m, n)
3     for (i = 0; i < m; i++) {
4        y[i] = 0.0;
5        for (j = 0; j < n; j++)
6           y[i] += A[i][j]*x[j];
7     }
```

If T_{serial} is the run-time of the serial program, and T_{parallel} is the run-time of the parallel program, recall that the *efficiency* E of the parallel program is the speedup S divided by the number of threads:

$$E = \frac{S}{t} = \frac{\left(\frac{T_{\text{serial}}}{T_{\text{parallel}}}\right)}{t} = \frac{T_{\text{serial}}}{t \times T_{\text{parallel}}}.$$

Table 5.6 Run-times and efficiencies of matrix-vector multiplication (times are in seconds).

	Matrix Dimension					
	8,000,000 × 8		8000 × 8000		8 × 8,000,000	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

Since $S \leq t$, $E \leq 1$. Table 5.6 shows the run-times and efficiencies of our matrix-vector multiplication with different sets of data and differing numbers of threads. In each case, the total number of floating point additions and multiplications is 64,000,000. An analysis that only considers arithmetic operations would predict that a single thread running the code would take the same amount of time for all three inputs. However, it's clear that this is *not* the case. The $8{,}000{,}000 \times 8$ system requires about 22% more time than the 8000×8000 system, and the $8 \times 8{,}000{,}000$ system requires about 26% more time than the 8000×8000 system. Both of these differences are at least partially attributable to cache performance.

Recall that a *write-miss* occurs when a core tries to update a variable that's not in cache, and it has to access main memory. A cache profiler (such as Valgrind [51]) shows that when the program is run with the $8{,}000{,}000 \times 8$ input, it has far more cache write-misses than either of the other inputs. The bulk of these occur in Line 4. Since the number of elements in the vector `y` is far greater in this case (8,000,000 vs. 8000 or 8), and each element must be initialized, it's not surprising that this line slows down the execution of the program with the $8{,}000{,}000 \times 8$ input.

Also recall that a *read-miss* occurs when a core tries to read a variable that's not in the cache, and it has to access main memory. A cache profiler shows that when the program is run with the $8 \times 8{,}000{,}000$ input, it has far more cache read-misses than either of the other inputs. These occur in Line 6, and a careful study of this program (see Exercise 5.12) shows that the main source of the differences is due to the reads of `x`. Once again, this isn't surprising, since for this input, `x` has 8,000,000 elements, versus only 8000 or 8 for the other inputs.

It should be noted that there may be other factors that are affecting the relative performance of the single-threaded program with the differing inputs. For example, we haven't taken into consideration whether virtual memory (see Subsection 2.2.4) has affected the performance of the program with the different inputs. How frequently does the CPU need to access the page table in main memory?

Of more interest to us, though, are the differences in efficiency as the number of threads is increased. The two-thread efficiency of the program with the $8 \times 8{,}000{,}000$ input is more than 20% less than the efficiency of the program with the $8{,}000{,}000 \times 8$ and the 8000×8000 inputs. The four-thread efficiency of the program with the $8 \times 8{,}000{,}000$ input is more than 50% less than the program's efficiency with the

8,000,000 × 8 and the 8000 × 8000 inputs. Why, then, is the multithreaded performance of the program so much worse with the 8 × 8,000,000 input?

In this case, once again, the answer has to do with cache. Let's take a look at the program when we run it with four threads. With the 8,000,000 × 8 input, `y` has 8,000,000 components, so each thread is assigned 2,000,000 components. With the 8000 × 8000 input, each thread is assigned 2000 components of `y`, and with the 8 × 8,000,000 input, each thread is assigned two components. On the system we used, a cache line is 64 bytes. Since the type of `y` is **double**, and a **double** is 8 bytes, a single cache line will store eight **double**s.

Cache coherence is enforced at the "cache-line level." That is, each time any value in a cache line is written, if the line is also stored in another core's cache, the entire *line* will be invalidated—not just the value that was written. The system we're using has two dual-core processors and each processor has its own cache. Suppose for the moment that threads 0 and 1 are assigned to one of the processors and threads 2 and 3 are assigned to the other. Also suppose that for the 8 × 8,000,000 problem all of `y` is stored in a single cache line. Then every write to some element of `y` will invalidate the line in the other processor's cache. For example, each time thread 0 updates `y[0]` in the statement

```
y[i] += A[i][j]*x[j];
```

if thread 2 or 3 is executing this code, it will have to reload `y`. Each thread will update each of its components 8,000,000 times. We see that with this assignment of threads to processors and components of `y` to cache lines, all the threads will have to reload `y` *many* times. This is going to happen in spite of the fact that only one thread accesses any one component of `y`—for example, only thread 0 accesses `y[0]`.

Each thread will update its assigned components of `y` a total of 16,000,000 times. It appears that many if not most of these updates are forcing the threads to access main memory. This is called **false sharing**. Suppose two threads with separate caches access different variables that belong to the same cache line. Further suppose at least one of the threads updates its variable. Then even though neither thread has written to a shared variable, the cache controller invalidates the entire cache line and forces the other threads to get the values of the variables from main memory. The threads aren't sharing anything (except a cache line), but the behavior of the threads with respect to memory access is the same as if they were sharing a variable, hence the name *false sharing*.

Why is false sharing not a problem with the other inputs? Let's look at what happens with the 8000 × 8000 input. Suppose thread 2 is assigned to one of the processors and thread 3 is assigned to another. (We don't actually know which threads are assigned to which processors, but it turns out—see Exercise 5.13—that it doesn't matter.) Thread 2 is responsible for computing

```
y[4000], y[4001], . . . , y[5999],
```

and thread 3 is responsible for computing

```
y[6000], y[6001], . . . , y[7999].
```

If a cache line contains eight consecutive **double**s, the only possibility for false sharing is on the interface between their assigned elements. If, for example, a single cache line contains

```
y[5996], y[5997], y[5998], y[5999],
y[6000], y[6001], y[6002], y[6003],
```

then it's conceivable that there might be false sharing of this cache line. However, thread 2 will access

```
y[5996], y[5997], y[5998], y[5999]
```

at the *end* of its **for** `i` loop, while thread 3 will access

```
y[6000], y[6001], y[6002], y[6003]
```

at the *beginning* of its iterations. So it's very likely that when thread 2 accesses, say, `y`[5996], thread 3 will be long done with all four of

```
y[6000], y[6001], y[6002], y[6003].
```

Similarly, when thread 3 accesses, say, `y`[6003], it's very likely that thread 2 won't be anywhere near starting to access

```
y[5996], y[5997], y[5998], y[5999].
```

It's therefore unlikely that false sharing of the elements of `y` will be a significant problem with the 8000×8000 input. Similar reasoning suggests that false sharing of `y` is unlikely to be a problem with the $8{,}000{,}000 \times 8$ input. Also note that we don't need to worry about false sharing of `A` or `x`, since their values are never updated by the matrix-vector multiplication code.

This brings up the question of how we might avoid false sharing in our matrix-vector multiplication program. One possible solution is to "pad" the `y` vector with dummy elements to ensure that any update by one thread won't affect another thread's cache line. Another alternative is to have each thread use its own private storage during the multiplication loop, and then update the shared storage when they're done. (See Exercise 5.15.)

5.10 Tasking

While many problems are straightforward to parallelize with OpenMP, they generally have a fixed or predetermined number of parallel blocks and loop iterations to schedule across participating threads. When this is not the case, the constructs we've seen

thus far make it difficult (or even impossible) to effectively parallelize the problem at hand. Consider, for instance, parallelizing a web server; HTTP requests may arrive at irregular times, and the server itself should ideally be able to respond to a potentially infinite number of requests. This is easy to conceptualize using a **while** loop, but recall our discussion in Section 5.5.1: **while** and **do**−**while** loops cannot be parallelized with OpenMP, nor can **for** loops that have an unbounded number of iterations. This poses potential issues for dynamic problems, including recursive algorithms, such as graph traversals, or producer-consumer style programs like web servers. To address these issues, OpenMP 3.0 introduced *Tasking* functionality [47]. Tasking has been successfully applied to a number of problems that were previously difficult to parallelize with OpenMP [1].

Tasking allows developers to specify independent units of computation with the `task` directive:

```
#pragma omp task
```

When a thread reaches a block of code with this directive, a new task is generated by the OpenMP run-time that will be scheduled for execution. It is important to note that the task will not necessarily be executed immediately, since there may be other tasks already pending execution. Task blocks behave similarly to a standard `parallel` region, but can launch an arbitrary number of tasks instead of only `num_threads`. In fact, tasks must be launched from within a `parallel` region but generally by only one of the threads in the team. Therefore a majority of programs that use Tasking functionality will contain an outer region that looks somewhat like:

```
#  pragma omp parallel
#  pragma omp single
   {
       ...
#      pragma omp task
       ...
   }
```

where the `parallel` directive creates a team of threads and the `single` directive instructs the runtime to only launch tasks from a single thread. If the `single` directive is omitted, subsequent `task` instances will be launched multiple times, one for each thread in the team.

To demonstrate OpenMP tasking functionality, recall our discussion on parallelizing the calculation of the first n Fibonacci numbers in Section 5.5.2. Due to the loop-carried dependence, results were unpredictable and, more importantly, often incorrect. However, we *can* parallelize this algorithm with the `task` directive. First, let's take a look at a recursive serial implementation that stores the sequence in a global array called `fibs`:

```
int fib(int n) {
    int i = 0;
    int j = 0;
```

```
    if (n <= 1) {
        // fibs is a global variable
        // It needs storage for n+1 ints
        fibs[n] = n;
        return n;
    }

    i = fib(n - 1);
    j = fib(n - 2);
    fibs[n] = i + j;
    return fibs[n];
}
```

This chain of recursive calls will be time-consuming, so let's execute each as a separate task that can run in parallel. We can do this by adding a `parallel` and a `single` directive before the initial (nonrecursive) call that starts `fib`, and adding **#pragma** omp task before each of the two recursive calls in `fib`. However, after we make this change, the results are incorrect—more specifically, except for `fib`[1], the sequence is all zeroes. This is because the default data scope for variables in tasks is private. So after completing each of the tasks

```
#  pragma omp task
i = fib(n - 1);
#  pragma omp task
j = fib(n - 2);
```

the results in `i` and `j` are lost: `i` and `j` retain their values from the initializations

```
int i = 0;
int j = 0;
```

at the beginning of the function. In other words, the memory locations that are assigned the results of `fib`(`n`−1) and `fib`(`n`−2) are not the same as the memory locations declared at the beginning of the function. So the values that are used to update `fibs`[`n`] are the zeroes assigned at the beginning of the function.

We can adjust the scope of `i` and `j` by declaring the variables to be `shared` in the tasks that execute the recursive call. However executing the program now will produce unpredictable results similar to our original attempt at parallelization. The problem here is that the order in which the various tasks execute isn't specified. In other words, our recursive function calls, `fib`(`n` − 1) and `fib`(`n` − 2) will be run eventually, but the thread executing the task that makes the recursive calls can continue to run and simply **return** the current value of `fibs`[`n`] early. We need to force this task to wait for its subtasks to complete with the `taskwait` directive, which operates as a `barrier` for tasks. We've put this all together in Program 5.6.

```
1  int fib(int n) {
2      int i = 0;
3      int j = 0;
4
5      if (n <= 1) {
6          fibs[n] = n;
7          return n;
8      }
9
10 #   pragma omp task shared(i)
11     i = fib(n - 1);
12
13 #   pragma omp task shared(j)
14     j = fib(n - 2);
15
16 #   pragma omp taskwait
17     fibs[n] = i + j;
18     return fibs[n];
19 }
```

Program 5.6: Computing the Fibonacci numbers using OpenMP tasks.

Our parallel Fibonacci program will now produce the correct results, but you may notice significant slowdowns with larger values of n; in fact, there is a good chance that the serial version of the program executes much faster! To gain an intuition as to why this occurs, recall our discussion of the overhead associated with forking and joining threads. Similarly, each task requires its own data environment to be generated upon creation, which takes time. There are a few options we can use to help reduce task creation overhead. The first option is to only create tasks in situations where n is large enough. We can do this with the **if** directive:

```
#pragma omp task shared(i) if(n > 20)
```

which in this case will restrict task creation to only occur when n is larger than 20 (chosen arbitrarily in this case based on some experimentation). Reviewing `fib` again, we can see that there will be a task executing `fib` itself, another executing `fib(n - 1)`, and a third executing `fib(n - 2)` for each recursive call. This is inefficient, because the parent task executing `fib` only launches two subtasks and then simply waits for their results. We can eliminate a task by having the parent thread perform one of the recursive calls to `fib` instead before doing the final calculation after the `taskwait` directive. On our 64-core testbed, these two changes halved the execution time of the program with $n = 35$.

While using the Tasking API requires a bit more planning and care to use—especially with data scoping and limiting runaway task creation—it allows a much broader set of problems to be parallelized by OpenMP.

5.11 Thread-safety[8]

Let's look at another potential problem that occurs in shared-memory programming: *thread-safety*. A block of code is **thread-safe** if it can be simultaneously executed by multiple threads without causing problems.

As an example, suppose we want to use multiple threads to "tokenize" a file. Let's suppose that the file consists of ordinary English text, and that the tokens are just contiguous sequences of characters separated from the rest of the text by white space—a space, a tab, or a newline. A simple approach to this problem is to divide the input file into lines of text and assign the lines to the threads in a round-robin fashion: the first line goes to thread 0, the second goes to thread 1, ..., the tth goes to thread t, the $t + 1$st goes to thread 0, and so on.

We'll read the text into an array of strings, with one line of text per string. Then we can use a `parallel` **for** directive with a `schedule(`**static**`,1)` clause to divide the lines among the threads.

One way to tokenize a line is to use the `strtok` function in `string.h`. It has the following prototype:

```
char* strtok(
      char*       string      /* in/out */,
      const char* separators  /* in     */);
```

Its usage is a little unusual: the first time it's called the `string` argument should be the text to be tokenized, so in our example it should be the line of input. For subsequent calls, the first argument should be `NULL`. The idea is that in the first call, `strtok` caches a pointer to `string`, and for subsequent calls it returns successive tokens taken from the cached copy. The characters that delimit tokens should be passed in `separators`, so we should pass in the string `" \t\n"` as the `separators` argument.

Given these assumptions, we can write the `Tokenize` function shown in Program 5.7. The main function has initialized the array `lines` so that it contains the input text, and `line_count` is the number of strings stored in `lines`. Although for our purposes, we only need the `lines` argument to be an input argument, the `strtok` function modifies its input. Thus when `Tokenize` returns, `lines` will be modified. When we run the program with a single thread, it correctly tokenizes the input stream. The first time we run it with two threads and the input

Pease porridge hot.
Pease porridge cold.
Pease porridge in the pot
Nine days old.

the output is also correct. However, the second time we run it with this input, we get the following output:

[8] This material is also covered in Chapter 4. So if you've already read that chapter, you may want to just skim this section.

```
1  void Tokenize(
2        char*  lines[]        /* in/out */,
3        int    line_count     /* in     */,
4        int    thread_count   /* in     */) {
5     int my_rank, i, j;
6     char *my_token;
7
8  #  pragma omp parallel num_threads(thread_count) \
9        default(none) private(my_rank, i, j, my_token) \
10       shared(lines, line_count)
11    {
12       my_rank = omp_get_thread_num();
13 #     pragma omp for schedule(static, 1)
14       for (i = 0; i < line_count; i++) {
15          printf("Thread %d > line %d = %s",
16                 my_rank, i, lines[i]);
17          j = 0;
18          my_token = strtok(lines[i], " \t\n");
19          while ( my_token != NULL ) {
20             printf("Thread %d > token %d = %s\n",
21                    my_rank, j, my_token);
22             my_token = strtok(NULL, " \t\n");
23             j++;
24          }
25       } /* for i */
26    }  /* omp parallel */
27
28 }  /* Tokenize */
```

Program 5.7: A first attempt at a multi-threaded tokenizer.

```
Thread 0 > line 0 = Pease porridge hot.
Thread 1 > line 1 = Pease porridge cold.
Thread 0 > token 0 = Pease
Thread 1 > token 0 = Pease
Thread 0 > token 1 = porridge
Thread 1 > token 1 = cold.
Thread 0 > line 2 = Pease porridge in the pot
Thread 1 > line 3 = Nine days old.
Thread 0 > token 0 = Pease
Thread 1 > token 0 = Nine
Thread 0 > token 1 = days
Thread 1 > token 1 = old.
```

What happened? Recall that `strtok` caches the input line. It does this by declaring a variable to have `static` storage class. This causes the value stored in this variable

to persist from one call to the next. Unfortunately for us, this cached string is shared, not private. Thus it appears that thread 1's call to `strtok` with the second line has apparently overwritten the contents of thread 0's call with the first line. Even worse, thread 0 has found a token ("days") that should be in thread 1's output.

The `strtok` function is therefore *not* thread-safe: if multiple threads call it simultaneously, the output it produces may not be correct. Regrettably, it's not uncommon for C library functions to fail to be thread-safe. For example, neither the random number generator `rand` in `stdlib.h` nor the time conversion function `localtime` in `time.h` is guaranteed to be thread-safe. In some cases, the C standard specifies an alternate, thread-safe version of a function. In fact, there is a thread-safe version of `strtok`:

```
char* strtok_r(
      char*       string      /* in/out */,
      const char* separators  /* in     */,
      char**      saveptr_p   /* in/out */);
```

The "_r" is supposed to suggest that the function is *re-entrant*, which is sometimes used as a synonym for thread-safe.[9] The first two arguments have the same purpose as the arguments to `strtok`. The `saveptr` argument is used by `strtok_r` for keeping track of where the function is in the input string; it serves the purpose of the cached pointer in `strtok`. We can correct our original `Tokenize` function by replacing the calls to `strtok` with calls to `strtok_r`. We simply need to declare a **char*** variable to pass in for the third argument, and replace the calls in line 18 and line 22 with the following calls:

```
my_token = strtok_r(lines[i], " \t\n", &saveptr);
. . .
my_token = strtok_r(NULL, " \t\n", &saveptr);
```

respectively.

5.11.1 Incorrect programs can produce correct output

Notice that our original version of the tokenizer program shows an especially insidious form of program error: The first time we ran it with two threads, the program produced correct output. It wasn't until a later run that we saw an error. This, unfortunately, is not a rare occurrence in parallel programs. It's especially common in shared-memory programs. Since, for the most part, the threads are running independently of each other, as we noted back at the beginning of the chapter, the exact

[9] However, the distinction is a bit more nuanced; being reentrant means a function can be interrupted and called again (reentered) in different parts of a program's control flow and still execute correctly. This can happen due to nested calls to the function or a trap/interrupt sent from the operating system. Since `strtok` uses a single static pointer to track its state while parsing, multiple calls to the function from different parts of a program's control flow will corrupt the string—therefore it is *not* reentrant. It's worth noting that although reentrant functions, such as `strtok_r`, can also be thread safe, there is no guarantee a reentrant function will *always* be thread safe—and vice versa. It's best to consult the documentation if there's any doubt.

CHAPTER

GPU programming with CUDA

6

6.1 GPUs and GPGPU

In the late 1990s and early 2000s, the computer industry responded to the demand for highly realistic computer video games and video animations by developing extremely powerful **graphics processing units** or **GPUs**. These processors, as their name suggests, are designed to improve the performance of programs that need to render many detailed images.

The existence of this computational power was a temptation to programmers who didn't specialize in computer graphics, and by the early 2000s they were trying to apply the power of GPUs to solving general computational problems, problems such as searching and sorting, rather than graphics. This became known as **General Purpose computing on GPUs** or **GPGPU**.

One of the biggest difficulties faced by the early developers of GPGPU was that the GPUs of the time could only be programmed using computer graphics APIs, such as Direct3D and OpenGL. So programmers needed to reformulate algorithms for general computational problems so that they used graphics concepts, such as vertices, triangles, and pixels. This added considerable complexity to the development of early GPGPU programs, and it wasn't long before several groups started work on developing languages and compilers that allowed programmers to implement general algorithms for GPUs in APIs that more closely resembled conventional, high-level languages for CPUs.

These efforts led to the development of several APIs for general purpose programming on GPUs. Currently the most widely used APIs are CUDA and OpenCL. CUDA was developed for use on Nvidia GPUs. OpenCL, on the other hand, was designed to be highly portable. It was designed for use on arbitrary GPUs *and* other processors—processors such as field programmable gate arrays (FPGAs) and digital signal processors (DSPs). To ensure this portability, an OpenCL program must include a good deal of code providing information about which systems it can be run on and information about how it should be run. Since CUDA was developed to run only on Nvidia GPUs, it requires relatively modest setup, and, as a consequence, we'll use it instead of OpenCL.

An Introduction to Parallel Programming. https://doi.org/10.1016/B978-0-12-804605-0.00013-0

Table 6.1 Execution of branch on a SIMD system.

Time	Datapaths with `x[i] >= 0`	Datapaths with `x[i] < 0`
1	Test `x[i] >= 0`	Test `x[i] >= 0`
2	`x[i] += 1`	Idle
3	Idle	`x[i] -= 2`

6.2 GPU architectures

As we've seen (see Chapter 2), CPU architectures can be extremely complex. However, we often think of a conventional CPU as a SISD device in Flynn's Taxonomy (see Section 2.3): the processor fetches an instruction from memory and executes the instruction on a small number of data items. The instruction is an element of the *Single Instruction stream*—the "SI" in SISD—and the data items are elements of the *Single Data stream*—the "SD" in SISD. GPUs, however, are composed of SIMD or *Single Instruction stream, Multiple Data stream* processors. So, to understand how to program them, we need to first look at their architecture.

Recall (from Section 2.3) that we can think of a SIMD processor as being composed of a single control unit and multiple datapaths. The control unit fetches an instruction from memory and broadcasts it to the datapaths. Each datapath either executes the instruction on *its* data or is idle.

For example, suppose there are n datapaths that share an n-element array `x`. Also suppose that the ith datapath will work with `x[i]`. Now suppose we want to add 1 to the nonnegative elements of `x` and subtract 2 from the negative elements of `x`. We might implement this with the following code:

```
/* Datapath i executes the following code */
if (x[i] >= 0)
   x[i] += 1;
else
   x[i] -= 2;
```

In a typical SIMD system, each datapath carries out the test `x[i] >= 0`. Then the datapaths for which the test is true execute `x[i] += 1`, while those for which `x[i] < 0` are *idle*. Then the roles of the datapaths are reversed: those for which `x[i] >= 0` are idle while the other datapaths execute `x[i] -= 2`. See Table 6.1.

A typical GPU can be thought of as being composed of one or more SIMD processors. Nvidia GPUs are composed of **Streaming Multiprocessors** or **SMs**.[1] One SM can have several control units and many more datapaths. So an SM can be thought of as consisting of one or more SIMD processors. The SMs, however, operate asynchronously: there is no penalty if one branch of an **if**–**else** executes on one SM, and

[1] The abbreviation that Nvidia uses for a streaming multiprocessor depends on the particular GPU microarchitecture. For example, Tesla and Fermi multiprocessors have SMs, Kepler multiprocessors have SMXs, and Maxwell multiprocessors have SMMs. More recent GPUs have SMs. We'll use SM, regardless of the microarchitecture.

Table 6.2 Execution of branch on a system with multiple SMs.

Time	Datapaths with `x[i] >= 0` (on SM A)	Datapaths with `x[i] < 0` (on SM B)
1	Test `x[i] >= 0`	Test `x[i] >= 0`
2	`x[i] += 1`	`x[i] -= 2`

the other executes on another SM. So in our preceding example, if all the threads with `x[i]` >= 0 were executing on one SM, and all the threads with `x[i]` < 0 were executing on another, the execution of our **if**−**else** example would require only two stages. (See Table 6.2.)

In Nvidia parlance, the datapaths are called cores, **Streaming Processors**, or **SPs**. Currently,[2] one of the most powerful Nvidia processor has 82 SMs, and each SM has 128 SPs for a total of 10,496 SPs. Since we use the term "core" to mean something else when we're discussing MIMD architectures, we'll use SP to denote an Nvidia datapath. Also note that Nvidia uses the term **SIMT** instead of SIMD. SIMT stands for Single Instruction Multiple Thread, and the term is used because threads on an SM that are executing the same instruction may not execute simultaneously: to hide memory access latency, some threads may block while memory is accessed and other threads, that have already accessed the data, may proceed with execution.

Each SM has a relatively small block of memory that is shared among its SPs. As we'll see, this memory can be accessed very quickly by the SPs. All of the SMs on a single chip also have access to a much larger block of memory that is shared among all the SPs. Accessing this memory is relatively slow. (See Fig. 6.1.)

The GPU and its associated memory are usually physically separate from the CPU and its associated memory. In Nvidia documentation, the CPU together with its associated memory is often called the **host**, and the GPU together with its memory is called the **device**. In earlier systems the physical separation of host and device memories required that data was usually explicitly transferred between CPU memory and GPU memory. That is, a function was called that would transfer a block of data from host memory to device memory or vice versa. So, for example, data read from a file by the CPU or output data generated by the GPU would have to be transferred between the host and device with an explicit function call. However, in more recent Nvidia systems (those with compute capability ≥ 3.0), the explicit transfers in the source code aren't needed for correctness, although they may be able to improve overall performance. (See Fig. 6.2.)

6.3 Heterogeneous computing

Up to now we've implicitly assumed that our parallel programs will be run on systems in which the individual processors have identical architectures. Writing a program

[2] Spring 2021.

6.6 Threads, blocks, and grids

You're probably wondering why we put a "1" in the angle brackets in our call to `Hello`:

```
Hello <<<1, thread_count>>>();
```

Recall that an Nvidia GPU consists of a collection of streaming multiprocessors (SMs), and each streaming multiprocessor consists of a collection of streaming processors (SPs). When a CUDA kernel runs, each individual thread will execute its code on an SP. With "1" as the first value in angle brackets, all of the threads that are started by the kernel call will run on a single SM. If our GPU has two SMs, we can try to use both of them with the kernel call

```
Hello <<<2, thread_count/2>>>();
```

If `thread_count` is even, this kernel call will start a total of `thread_count` threads, and the threads will be divided between the two SMs: `thread_count`/2 threads will run on each SM. (What happens if `thread_count` is odd?)

CUDA organizes threads into blocks and grids. A **thread block** (or just a **block** if the context makes it clear) is a collection of threads that run on a single SM. In a kernel call the first value in the angle brackets specifies the number of thread blocks. The second value is the number of threads in each thread block. So when we started the kernel with

```
Hello <<<1, thread_count>>>();
```

we were using one thread block, which consisted of `thread_count` threads, and, as a consequence, we only used one SM.

We can modify our greetings program so that it uses a user-specified number of blocks, each consisting of a user-specified number of threads. (See Program 6.2.) In this program we get both the number of thread blocks and the number of threads in each block from the command line. Now the kernel call starts `blk_ct` thread blocks, each of which contains `th_per_blk` threads.

When the kernel is started, each block is assigned to an SM, and the threads in the block are then run on that SM. The output is similar to the output from the original program, except that now we're using two system-defined variables: `threadIdx.x` and `blockIdx.x`. As you've probably guessed, `threadIdx.x` gives a thread's rank or index in its block, and `blockIdx.x` gives a block's rank in the *grid*.

A **grid** is just the collection of thread blocks started by a kernel. So a thread block is composed of threads, and a grid is composed of thread blocks.

There are several built-in variables that a thread can use to get information on the grid started by the kernel. The following four variables are structs that are initialized in each thread's memory when a kernel begins execution:

- `threadIdx`: the rank or index of the thread in its thread block.
- `blockDim`: the dimensions, shape, or size of the thread blocks.

```
1  #include <stdio.h>
2  #include <cuda.h>   /* Header file for CUDA */
3
4  /* Device code:  runs on GPU */
5  __global__ void Hello(void) {
6
7     printf("Hello from thread %d in block %d\n",
8           threadIdx.x, blockIdx.x);
9  }  /* Hello */
10
11
12 /* Host code:  Runs on CPU */
13 int main(int argc, char* argv[]) {
14    int blk_ct;                 /* Number of thread blocks */
15    int th_per_blk;      /* Number of threads in each block */
16
17    blk_ct = strtol(argv[1], NULL, 10);
18                   /* Get number of blocks from command line */
19    th_per_blk = strtol(argv[2], NULL, 10);
20        /* Get number of threads per block from command line */
21
22    Hello <<<blk_ct, th_per_blk>>>();
23             /* Start blk_ct*th_per_blk threads on GPU, */
24
25    cudaDeviceSynchronize();        /* Wait for GPU to finish */
26
27    return 0;
28 }  /* main */
```

Program 6.2: CUDA program that prints greetings from threads in multiple blocks.

- `blockIdx`: the rank or index of the block within the grid.
- `gridDim`: the dimensions, shape, or size of the grid.

All of these structs have three fields, `x`, `y`, and `z`,[5] and the fields all have unsigned integer types. The fields are often convenient for applications. For example, an application that uses graphics may find it convenient to assign a thread to a point in two- or three-dimensional space, and the fields in `threadIdx` can be used to indicate the point's position. An application that makes extensive use of matrices may find it convenient to assign a thread to an element of a matrix, and the fields in `threadIdx` can be used to indicate the column and row of the element.

[5] Nvidia devices that have compute capability < 2 (see Section 6.7) only allow x- and y-dimensions in a grid.

When we call a kernel with something like

```
int blk_ct, th_per_blk;
...
Hello <<<blk_ct, th_per_blk>>>();
```

the three-element structures `gridDim` and `blockDim` are initialized by assigning the values in angle brackets to the `x` fields. So, effectively, the following assignments are made:

```
gridDim.x = blk_ct;
blockDim.x = th_per_blk;
```

The `y` and `z` fields are initialized to 1. If we want to use values other than 1 for the `y` and `z` fields, we should declare two variables of type `dim3`, and pass them into the call to the kernel. For example,

```
dim3 grid_dims, block_dims;
grid_dims.x = 2;
grid_dims.y = 3;
grid_dims.z = 1;
block_dims.x = 4;
block_dims.y = 4;
block_dims.z = 4;
...
Kernel <<<grid_dims, block_dims>>> (...);
```

This should start a grid with $2 \times 3 \times 1 = 6$ blocks, each of which has $4^3 = 64$ threads.

Note that all the blocks must have the same dimensions. More importantly, CUDA requires that thread blocks be independent. So one thread block must be able to complete its execution, regardless of the states of the other thread blocks: the thread blocks can be executed sequentially in any order, or they can be executed in parallel. This ensures that the GPU can schedule a block to execute solely on the basis of the state of that block: it doesn't need to check on the state of any other block.[6]

6.7 Nvidia compute capabilities and device architectures[7]

There are limits on the number of threads and the number of blocks. The limits depend on what Nvidia calls the **compute capability** of the GPU. The compute capability is a number having the form $a.b$. Currently the a-value or major revision number can be 1, 2, 3, 5, 6, 7, 8. (There is no major revision number 4.) The possible b-values or minor revision numbers depend on the major revision value, but currently

[6] With the introduction of CUDA 9 and the Pascal processor, it became possible to synchronize threads in multiple blocks. See Subsection 7.1.13 and Exercise 7.6.

[7] The values in this section are current as of spring 2021, but some of them may change when Nvidia releases new GPUs and new versions of CUDA.

Table 6.3 GPU architectures and compute capabilities.

Name	Ampere	Tesla	Fermi	Kepler	Maxwell	Pascal	Volta	Turing
Compute capability	8.0	1.*b*	2.*b*	3.*b*	5.*b*	6.*b*	7.0	7.5

they fall in the range 0–7. CUDA no longer supports devices with compute capability < 3.

For devices with compute capability > 1, the maximum number of threads per block is 1024. For devices with compute capability 2.*b*, the maximum number of threads that can be assigned to a single SM is 1536, and for devices with compute capability > 2, the maximum is currently 2048. There are also limits on the sizes of the dimensions in both blocks and grids. For example, for compute capability > 1, the maximum `x`- or `y`-dimension is 1024, and the maximum `z`-dimension is 64. For further information, see the appendix on compute capabilities in the CUDA C++ Programming Guide [11].

Nvidia also has names for the microarchitectures of its GPUs. Table 6.3 shows the current list of architectures and some of their corresponding compute capabilities. Somewhat confusingly, Nvidia also uses Tesla as the name for their products targeting GPGPU.

We should note that Nvidia has a number of "product families" that can consist of anything from an Nvidia-based graphics card to a "system on a chip," which has the main hardware components of a system, such as a mobile phone in a single integrated circuit.

Finally, note that there are a number of versions of the CUDA API, and they do *not* correspond to the compute capabilities of the different GPUs.

6.8 Vector addition

GPUs and CUDA are designed to be especially effective when they run data-parallel programs. So let's write a very simple, data-parallel CUDA program that's embarrassingly parallel: a program that adds two vectors or arrays. We'll define three n-element arrays, `x`, `y`, and `z`. We'll initialize `x` and `y` on the host. Then a kernel can start at least n threads, and the ith thread will add

```
z[i] = x[i] + y[i];
```

Since GPUs tend to have more 32-bit than 64-bit floating point units, let's use arrays of `floats` rather than `doubles`:

```
float *x, *y, *z;
```

After allocating and initializing the arrays, we'll call the kernel, and after the kernel completes execution, the program checks the result, frees memory, and quits. See Program 6.3, which shows the kernel and the main function.

Let's take a closer look at the program.

```
1  __global__ void Vec_add(
2        const float x[]  /* in  */,
3        const float y[]  /* in  */,
4        float       z[]  /* out */,
5        const int   n    /* in  */) {
6     int my_elt = blockDim.x * blockIdx.x + threadIdx.x;
7
8     /* total threads = blk_ct*th_per_blk may be > n */
9     if (my_elt < n)
10       z[my_elt] = x[my_elt] + y[my_elt];
11 }  /* Vec_add */
12
13 int main(int argc, char* argv[]) {
14    int n, th_per_blk, blk_ct;
15    char i_g;  /* Are x and y user input or random? */
16    float *x, *y, *z, *cz;
17    double diff_norm;
18
19    /* Get the command line arguments, and set up vectors */
20    Get_args(argc, argv, &n, &blk_ct, &th_per_blk, &i_g);
21    Allocate_vectors(&x, &y, &z, &cz, n);
22    Init_vectors(x, y, n, i_g);
23
24    /* Invoke kernel and wait for it to complete          */
25    Vec_add <<<blk_ct, th_per_blk>>>(x, y, z, n);
26    cudaDeviceSynchronize();
27
28    /* Check for correctness */
29    Serial_vec_add(x, y, cz, n);
30    diff_norm = Two_norm_diff(z, cz, n);
31    printf("Two-norm of difference between host and ");
32    printf("device = %e\n", diff_norm);
33
34    /* Free storage and quit */
35    Free_vectors(x, y, z, cz);
36    return 0;
37 }  /* main */
```

Program 6.3: Kernel and `main` function of a CUDA program that adds two vectors.

6.8.1 The kernel

In the kernel (Lines 1–11), we first determine which element of `z` the thread should compute. We've chosen to make this index the same as the *global* rank or index of the thread. Since we're only using the `x` fields of the `blockDim` and `threadIdx` structs, there are a total of

Table 6.4 Global thread ranks or indexes in a grid with 4 blocks and 5 threads per block.

	threadIdx.x				
blockIdx.x	**0**	**1**	**2**	**3**	**4**
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14
3	15	16	17	18	19

```
gridDim.x * blockDim.x
```

threads. So we can assign a unique "global" rank or index to each thread by using the formula

```
rank = blockDim.x * blockIdx.x + threadIdx.x
```

For example, if we have four blocks and five threads in each block, then the global ranks or indexes are shown in Table 6.4. In the kernel, we assign this global rank to `my_elt` and use this as the subscript for accessing each thread's elements of the arrays `x`, `y`, and `z`.

Note that we've allowed for the possibility that the total number of threads may not be exactly the same as the number of components of the vectors. So before carrying out the addition,

```
z[my_elt] = x[my_elt] + y[my_elt];
```

we first check that `my_elt` < `n`. For example, if we have $n = 997$, and we want at least two blocks with at least two threads per block, then, since 997 is prime, we can't possibly have exactly 997 threads. Since this kernel needs to be executed by at least n threads, we must start more than 997. For example, we might use four blocks of 256 threads, and the last 27 threads in the last block would skip the line

```
z[my_elt] = x[my_elt] + y[my_elt];
```

Note that if we needed to run our program on a system that didn't support CUDA, we could replace the kernel with a serial vector addition function. (See Program 6.4.) So we can view the CUDA kernel as taking the serial **for** loop and assigning each iteration to a different thread. This is often how we start the design process when we want to parallelize a serial code for CUDA: assign the iterations of a loop to individual threads.

Also note that if we apply Foster's method to parallelizing the serial vector sum, and we make the tasks the additions of the individual components, then we don't need to do anything for the communication and aggregation phases, and the mapping phase simply assigns each addition to a thread.

```
1  void Serial_vec_add(
2        const float  x[]   /* in  */,
3        const float  y[]   /* in  */,
4        float        cz[]  /* out */,
5        const int    n     /* in  */) {
6
7     for (int i = 0; i < n; i++)
8        cz[i] = x[i] + y[i];
9  } /* Serial_vec_add */
```

Program 6.4: Serial vector addition function.

6.8.2 Get_args

After declaring the variables, the `main` function calls a `Get_args` function, which returns n, the number of elements in the arrays, `blk_ct`, the number of thread blocks, and `th_per_blk`, the number of threads in each block. It gets these from the command line. It also returns a **char** `i_g`. This tells the program whether the user will *i*nput `x` and `y` or whether it should *g*enerate them using a random number generator. If the user doesn't enter the correct number of command line arguments, the function prints a usage summary and terminates execution. Also if n is greater than the total number of threads, it prints a message and terminates. (See Program 6.5.) Note that `Get_args` is written in standard C, and it runs completely on the host.

6.8.3 Allocate_vectors and managed memory

After getting the command line arguments, the main function calls `Allocate_vectors`, which allocates storage for four n-element arrays of **float**:

```
x, y, z, cz
```

The first three arrays are used on both the host and the device. The fourth array, `cz`, is only used on the host: we use it to compute the vector sum with one core of the host. We do this so that we can check the result computed on the device. (See Program 6.6.)

First note that since `cz` is only used on the host, we allocate its storage using the standard C library function `malloc`. For the other three arrays, we allocate storage in Lines 9–11 using the CUDA function

```
__host__ cudaError_t cudaMallocManaged (
            void**      devPtr   /* out */,
            size_t      size     /* in  */,
            unsigned    flags    /* in  */);
```

The `__host__` qualifier is a CUDA addition to C, and it indicates that the function should be called and run on the host. This is the default for functions in CUDA

```
1  void Get_args(
2        const int argc            /* in  */,
3        char*     argv[]          /* in  */,
4        int*      n_p             /* out */,
5        int*      blk_ct_p        /* out */,
6        int*      th_per_blk_p    /* out */,
7        char*     i_g             /* out */) {
8     if (argc != 5) {
9        /* Print an error message and exit */
10       ...
11    }
12
13    *n_p = strtol(argv[1], NULL, 10);
14    *blk_ct_p = strtol(argv[2], NULL, 10);
15    *th_per_blk_p = strtol(argv[3], NULL, 10);
16    *i_g = argv[4][0];
17
18    /* Is n > total thread count = blk_ct*th_per_blk? */
19    if (*n_p > (*blk_ct_p)*(*th_per_blk_p)) {
20       /* Print an error message and exit */
21       ...
22    }
23 }  /* Get_args */
```

Program 6.5: `Get_args` function from CUDA program that adds two vectors.

```
1  void Allocate_vectors(
2        float** x_p     /* out */,
3        float** y_p     /* out */,
4        float** z_p     /* out */,
5        float** cz_p    /* out */,
6        int     n       /* in  */) {
7
8     /* x, y, and z are used on host and device */
9     cudaMallocManaged(x_p, n*sizeof(float));
10    cudaMallocManaged(y_p, n*sizeof(float));
11    cudaMallocManaged(z_p, n*sizeof(float));
12
13    /* cz is only used on host */
14    *cz_p = (float*) malloc(n*sizeof(float));
15 }  /* Allocate_vectors */
```

Program 6.6: Array allocation function of CUDA program that adds two vectors.

programs, so it can be omitted when we're writing our own functions, and they'll only be run on the host.

The return value, which has type `cudaError_t`, allows the function to return an error. Most CUDA functions return a `cudaError_t` value, and if you're having problems with your code, it is a very good idea to check it. However, always checking it tends to clutter the code, and this can distract us from the main purpose of a program. So in the code we discuss we'll generally ignore `cudaError_t` return values.

The first argument is a pointer to a pointer: it refers to the pointer that's being allocated. The second argument specifies the number of bytes that should be allocated. The `flags` argument controls which kernels can access the allocated memory. It defaults to `cudaMemAttachGlobal` and can be omitted.

The function `cudaMallocManaged` is one of several CUDA memory allocation functions. It allocates memory that will be automatically managed by the "unified memory system." This is a relatively recent addition to CUDA,[8] and it allows a programmer to write CUDA programs as if the host and device shared a single memory: pointers referring to memory allocated with `cudaMallocManaged` can be used on both the device and the host, even when the host and the device have separate physical memories. As you can imagine this greatly simplifies programming, but there are some cautions. Here are a few:

1. Unified memory requires a device with compute capability ≥ 3.0, and a 64-bit host operating system.
2. On devices with compute capability < 6.0 memory allocated with `cudaMallocManaged` cannot be simultaneously accessed by both the device and the host. When a kernel is executing, it has exclusive access to memory allocated with `cudaMallocManaged`.
3. Kernels that use unified memory can be slower than kernels that treat device memory as separate from host memory.

The last caution has to do with the transfer of data between the host and the device. When a program uses unified memory, it is up to the system to decide when to transfer from the host to the device or vice versa. In programs that explicitly transfer data, it is up to the programmer to include code that implements the transfers, and she may be able to exploit her knowledge of the code to do things that reduce the cost of transfers, things such as omitting some transfers or overlapping data transfer with computation.

At the end of this section we'll briefly discuss the modifications required if you want to explicitly handle the transfers between host and device.

6.8.4 Other functions called from `main`

Except for the `Free_vectors` function, the other host functions that we call from `main` are just standard C.

[8] It first became available in CUDA 6.0.

The function `Init_vectors` either reads `x` and `y` from `stdin` using `scanf` or generates them using the C library function `random`. It uses the last command line argument `i_g` to decide which it should do.

The `Serial_vec_add` function (Program 6.4) just adds `x` and `y` on the host using a **for** loop. It stores the result in the host array `cz`.

The `Two_norm_diff` function computes the "distance" between the vector `z` computed by the kernel and the vector `cz` computed by `Serial_vec_add`. So it takes the difference between corresponding components of `z` and `cz`, squares them, adds the squares, and takes the square root:

$$\sqrt{(z[0] - cz[0])^2 + (z[1] - cz[1])^2 + \cdots + (z[n-1] - cz[n-1])^2}.$$

See Program 6.7.

```
1  double Two_norm_diff(
2        const float  z[]   /* in */,
3        const float  cz[]  /* in */,
4        const int    n     /* in */) {
5     double diff, sum = 0.0;
6
7     for (int i = 0; i < n; i++) {
8        diff = z[i] - cz[i];
9        sum += diff*diff;
10    }
11    return sqrt(sum);
12 }  /* Two_norm_diff */
```

Program 6.7: C function that finds the distance between two vectors.

The `Free_vectors` function just frees the arrays allocated by `Allocate_vectors`. The array `cz` is freed using the C library function `free`, but since the other arrays are allocated using `cudaMallocManaged`, they must be freed by calling `cudaFree`:

```
__host__ __device__ cudaError_t cudaFree ( void* ptr )
```

The qualifier `__device__` is a CUDA addition to C, and it indicates that the function can be called from the device. So `cudaFree` can be called from the host or the device. However, if a pointer is allocated on the device, it cannot be freed on the host, and vice versa.

It's important to note that unless memory allocated on the device is explicitly freed by the program, it won't be freed until the program terminates. So if a CUDA program calls two (or more) kernels, and the memory used by the first kernel isn't explicitly freed before the second is called, it will remain allocated, regardless of whether the second kernel actually uses it.

See Program 6.8.

```
1  void Free_vectors(
2        float* x    /* in/out */,
3        float* y    /* in/out */,
4        float* z    /* in/out */,
5        float* cz   /* in/out */) {
6
7     /* Allocated with cudaMallocManaged */
8     cudaFree(x);
9     cudaFree(y);
10    cudaFree(z);
11
12    /* Allocated with malloc */
13    free(cz);
14 } /* Free_vectors */
```

Program 6.8: CUDA function that frees four arrays.

6.8.5 Explicit memory transfers[9]

Let's take a look at how to modify the vector addition program for a system that doesn't provide unified memory. Program 6.9 shows the kernel and main function for the modified program.

The first thing to notice is that the kernel is unchanged: the arguments are `x`, `y`, `z`, and `n`. It finds the thread's global index, `my_elt`, and if this is less than `n`, it adds the elements of `x` and `y` to get the corresponding element of `z`.

The basic structure of the `main` function is almost the same. However, since we're assuming unified memory is unavailable, pointers on the host aren't valid on the device, and vice versa: an address on the host may be illegal on the device, or, even worse, it might refer to memory that the device is using for some other purpose. Similar problems occur if we try to use a device address on the host. So instead of declaring and allocating storage for three arrays that are all valid on both the host and the device, we declare and allocate storage for three arrays that are valid on the host `hx`, `hy`, and `hz`, *and* we declare and allocate storage for three arrays that are valid on the device, `dx`, `dy`, and `dz`. The declarations are in Lines 15–16, and the allocations are in the `Allocate_vectors` function called in Line 20. The function itself is in Program 6.10. Since unified memory isn't available, instead of using `cudaMallocManaged`, we use the C library function `malloc` for the host arrays, and the CUDA function `cudaMalloc` for the device arrays:

```
__host__ __device__ cudaError_t cudaMalloc (
            void** dev_p   /* out */,
            size_t size    /* in  */);
```

[9] If your device has compute capability **≥ 3.0**, you can skip this section.

```
1  __global__ void Vec_add(
2        const float  x[]  /* in  */,
3        const float  y[]  /* in  */,
4        float        z[]  /* out */,
5        const int    n    /* in  */) {
6     int my_elt = blockDim.x * blockIdx.x + threadIdx.x;
7
8     if (my_elt < n)
9        z[my_elt] = x[my_elt] + y[my_elt];
10 }  /* Vec_add */
11
12 int main(int argc, char* argv[]) {
13    int n, th_per_blk, blk_ct;
14    char i_g;  /* Are x and y user input or random? */
15    float *hx, *hy, *hz, *cz; /* Host arrays         */
16    float *dx, *dy, *dz;      /* Device arrays       */
17    double diff_norm;
18
19    Get_args(argc, argv, &n, &blk_ct, &th_per_blk, &i_g);
20    Allocate_vectors(&hx, &hy, &hz, &cz, &dx, &dy, &dz, n);
21    Init_vectors(hx, hy, n, i_g);
22
23    /* Copy vectors x and y from host to device */
24    cudaMemcpy(dx, hx, n*sizeof(float), cudaMemcpyHostToDevice);
25    cudaMemcpy(dy, hy, n*sizeof(float), cudaMemcpyHostToDevice);
26
27
28    Vec_add <<<blk_ct, th_per_blk>>>(dx, dy, dz, n);
29
30    /* Wait for kernel to complete and copy result to host */
31    cudaMemcpy(hz, dz, n*sizeof(float), cudaMemcpyDeviceToHost);
32
33    Serial_vec_add(hx, hy, cz, n);
34    diff_norm = Two_norm_diff(hz, cz, n);
35    printf("Two-norm of difference between host and ");
36    printf("device = %e\n", diff_norm);
37
38    Free_vectors(hx, hy, hz, cz, dx, dy, dz);
39
40    return 0;
41 }  /* main */
```

Program 6.9: Part of CUDA program that implements vector addition without unified memory.

```
1  void Allocate_vectors(
2        float** hx_p  /* out */,
3        float** hy_p  /* out */,
4        float** hz_p  /* out */,
5        float** cz_p  /* out */,
6        float** dx_p  /* out */,
7        float** dy_p  /* out */,
8        float** dz_p  /* out */,
9        int     n     /* in  */)  {
10
11    /* dx, dy, and dz are used on device */
12    cudaMalloc(dx_p, n*sizeof(float));
13    cudaMalloc(dy_p, n*sizeof(float));
14    cudaMalloc(dz_p, n*sizeof(float));
15
16    /* hx, hy, hz, cz are used on host */
17    *hx_p = (float*) malloc(n*sizeof(float));
18    *hy_p = (float*) malloc(n*sizeof(float));
19    *hz_p = (float*) malloc(n*sizeof(float));
20    *cz_p = (float*) malloc(n*sizeof(float));
21 }  /* Allocate_vectors */
```

Program 6.10: `Allocate_vectors` function for CUDA vector addition program that doesn't use unified memory.

The first argument is a reference to a pointer that will be used *on the device*. The second argument specifies the number of bytes to allocate on the device.

After we've initialized `hx` and `hy` on the host, we copy their contents over to the device, storing the transferred contents in the memory allocated for `dx` and `dy`, respectively. The copying is done in Lines 24–26 using the CUDA function `cudaMemcpy`:

```
__host__ cudaError_t cudaMemcpy (
             void*          dest    /* out */,
             const void*    source  /* in  */,
             size_t         count   /* in  */,
             cudaMemcpyKind kind    /* in  */);
```

This copies `count` bytes from the memory referred to by `source` into the memory referred to by `dest`. The type of the `kind` argument, `cudaMemcpyKind`, is an enumerated type defined by CUDA that specifies where the `source` and `dest` pointers are located. For our purposes the two values of interest are `cudaMemcpyHostToDevice` and `cudaMemcpyDeviceToHost`. The first indicates that we're copying from the host to the device, and the second indicates that we're copying from the device to the host.

The call to the kernel in Line 28 uses the pointers `dx`, `dy`, and `dz`, because these are addresses that are valid on the *device*.

After the call to the kernel, we copy the result of the vector addition from the device to the host in Line 31 using `cudaMemcpy` again. A call to `cudaMemcpy` is *synchronous*, so it waits for the kernel to finish executing before carrying out the transfer. So in this version of vector addition we do *not* need to use `cudaDeviceSynchronize` to ensure that the kernel has completed before proceeding.

After copying the result from the device back to the host, the program checks the result, frees the memory allocated on the host and the device, and terminates. So for this part of the program, the only difference from the original program is that we're freeing seven pointers instead of four. As before, the `Free_vectors` function frees the storage allocated on the host with the C library function `free`. It uses `cudaFree` to free the storage allocated on the device.

6.9 Returning results from CUDA kernels

There are several things that you should be aware of regarding CUDA kernels. First, they always have return type **void**, so they can't be used to return a value. They also can't return anything to the host through the standard C pass-by-reference. The reason for this is that addresses on the host are, in most systems, invalid on the device, and vice versa. For example, suppose we try something like this:

```
__global__ void Add(int x, int y, int* sum_p) {
   *sum_p = x + y;
}  /*  Add */

int main(void) {
   int sum = -5;
   Add <<<1, 1>>> (2, 3, &sum);
   cudaDeviceSynchronize();
   printf("The sum is %d\n", sum);

   return 0;
}
```

It's likely that either the host will print -5 or the device will hang. The reason is that the address `&sum` is probably invalid on the device. So the dereference

```
   *sum_p = x + y;
```

is attempting to assign `x + y` to an invalid memory location.

There are several possible approaches to "returning" a result to the host from a kernel. One is to declare pointer variables and allocate a single memory location. On a system that supports unified memory, the computed value will be automatically copied back to host memory:

```
__global__ void Add(int x, int y, int* sum_p) {
   *sum_p = x + y;
}  /*  Add */
```

```
int main(void) {
   int* sum_p;
   cudaMallocManaged(&sum_p, sizeof(int));
   *sum_p = -5;
   Add <<<1, 1>>> (2, 3, sum_p);
   cudaDeviceSynchronize();
   printf("The sum is %d\n", *sum_p);
   cudaFree(sum_p);

   return 0;
}
```

If your system doesn't support unified memory, the same idea will work, but the result will have to be explicitly copied from the device to the host:

```
__global__ void Add(int x, int y, int *sum_p) {
   *sum_p = x + y;
}  /*  Add */

int main(void) {
   int *hsum_p, *dsum_p;
   hsum_p = (int*) malloc(sizeof(int));
   cudaMalloc(&dsum_p, sizeof(int));
   *hsum_p = -5;
   Add <<<1, 1>>> (2, 3, dsum_p);
   cudaMemcpy(hsum_p, dsum_p, sizeof(int),
         cudaMemcpyDeviceToHost);
   printf("The sum is %d\n", *hsum_p);
   free(hsum_p);
   cudaFree(dsum_p);

   return 0;
}
```

Note that in both the unified and non-unified memory settings, we're returning a *single* value from the device to the host.

If unified memory is available, another option is to use a global managed variable for the sum:

```
__managed__ int sum;

__global__ void Add(int x, int y) {
   sum = x + y;
}  /*  Add */

int main(void) {
   sum = -5;
   Add <<<1, 1>>> (2, 3);
```

```
    cudaDeviceSynchronize();
    printf("After kernel:  The sum is %d\n", sum);

    return 0;
}
```

The qualifier `__managed__` declares `sum` to be a managed **int** that is accessible to all the functions, regardless of whether they run on the host or the device. Since it's managed, the same restrictions apply to it that apply to managed variables allocated with `cudaMallocManaged`. So this option is unavailable on systems with compute capability < 3.0, and on systems with compute capability < 6.0, `sum` can't be accessed on the host while the kernel is running. So after the call to `Add` has started, the host can't access `sum` until after the call to `cudaDeviceSynchronize` has completed.

Since this last approach uses a global variable, it has the usual problem of reduced modularity associated with global variables.

6.10 CUDA trapezoidal rule I

6.10.1 The trapezoidal rule

Let's try to implement a CUDA version of the trapezoidal rule. Recall (see Section 3.2.1) that the trapezoidal rule estimates the area between an interval on the x-axis and the graph of a function by dividing the interval into subintervals and approximating the area between each subinterval and the graph by the area of a trapezoid. (See Fig. 3.3.) So if the interval is $[a, b]$ and there are n trapezoids, we'll divide $[a, b]$ into n equal subintervals, and the length of each subinterval will be

$$h = (b - a)/n.$$

Then if x_i is the left end point of the ith subinterval,

$$x_i = a + ih,$$

for $i = 0, 1, 2, \ldots, n - 1$. To simplify the notation, we'll also denote b, the right end point of the interval, as

$$b = x_n = a + nh.$$

Recall that if a trapezoid has height h and base lengths c and d, then its area is

$$\frac{h}{2}(c + d).$$

So if we think of the length of the subinterval $[x_i, x_{i+1}]$ as the height of the ith trapezoid, and $f(x_i)$ and $f(x_{i+1})$ as the two base lengths (see Fig. 3.4), then the area of the ith trapezoid is

$$\frac{h}{2}\left[f(x_i) + f(x_{i+1})\right].$$

This gives us a total approximation of the area between the graph and the x-axis as

$$\frac{h}{2}[f(x_0)+f(x_1)]+\frac{h}{2}[f(x_1)+f(x_2)]+\cdots+\frac{h}{2}[f(x_{n-1})+f(x_n)],$$

and we can rewrite this as

$$h\left[\frac{1}{2}(f(a)+f(b))+(f(x_1)+f(x_2)+\cdots+f(x_{n-1}))\right].$$

We can implement this with the serial function shown in Program 6.11.

```
1  float Serial_trap(
2        const float  a  /* in */,
3        const float  b  /* in */,
4        const int    n  /* in */) {
5     float x, h = (b-a)/n;
6     float trap = 0.5*(f(a) + f(b));
7
8     for (int i = 1; i <= n-1; i++) {
9        x = a + i*h;
10       trap += f(x);
11    }
12    trap = trap*h;
13
14    return trap;
15 }  /* Serial_trap */
```

Program 6.11: A serial function implementing the trapezoidal rule for a single CPU.

6.10.2 A CUDA implementation

If n is large, the vast majority of the work in the serial implementation is done by the **for** loop. So when we apply Foster's method to the trapezoidal rule, we're mainly interested in two types of tasks: the first is the evaluation of the function f at x_i, and the second is the addition of $f(x_i)$ into `trap`. Here $i = 1, \ldots, n-1$. The second type of task depends on the first. So we can aggregate these two tasks.

This suggests that each thread in our CUDA implementation might carry out one iteration of the serial **for** loop. We can assign a unique integer rank to each thread as we did with the vector addition program. Then we can compute an x-value, the function value, and add in the function value to the "running sum":

```
/* h and trap are formal arguments to the kernel */
int my_i = blockDim.x * blockIdx.x + threadIdx.x;
float my_x = a + my_i*h;
float my_trap = f(my_x);
float trap += my_trap;
```

However, it's immediately obvious that there are several problems here:

1. We haven't initialized `h` or `trap`.
2. The `my_i` value can be too large or too small: the serial loop ranges from 1 up to and including $n - 1$. The smallest value for `my_i` is 0 and the largest is the total number of threads minus 1.
3. The variable `trap` must be shared among the threads. So the addition of `my_trap` forms a race condition: when multiple threads try to update `trap` at roughly the same time, one thread can overwrite another thread's result, and the final value in trap may be wrong. (For a discussion of race conditions, see Section 2.4.3.)
4. The variable `trap` in the serial code is returned by the function, and, as we've seen, kernels must have **void** return type.
5. We see from the serial code that we need to multiply the total in `trap` by `h` after all of the threads have added their results.

Program 6.12 shows how we might deal with these problems. In the following sections, we'll look at the rationales for the various choices we've made.

6.10.3 Initialization, return value, and final update

To deal with the initialization and the final update (Items 1 and 5), we could try to select a single thread—say, thread 0 in block 0—to carry out the operations:

```
int my_i = blockDim.x * blockIdx.x + threadIdx.x;
if (my_i == 0) {
   h = (b-a)/n;
   trap = 0.5*(f(a) + f(b));
}
...
if (my_i == 0)
   trap = trap*h;
```

There are (at least) a couple of problems with these options: formal arguments to functions are private to the executing thread and **thread synchronization**.

Kernel and function arguments are private to the executing thread.

Like the threads started in Pthreads and OpenMP, each CUDA thread has its own stack and, and since formal arguments are allocated on the thread's stack, each thread has its own private variables `h` and `trap`. So any changes made to one of these variables by one thread won't be visible to the other threads. We could have each thread initialize `h`, but we could also just do the initialization once in the host. If we do this before the kernel is called, each thread will get a copy of the value of `h`.

Things are more complicated with `trap`. Since it's updated by multiple threads, it must be *shared* among the threads. We can achieve the *effect* of sharing `trap` by allocating storage for a memory location before the kernel is called. This allocated memory location will correspond to what we've been calling `trap`. Now we can pass

```
1  __global__ void Dev_trap(
2        const float  a        /* in      */,
3        const float  b        /* in      */,
4        const float  h        /* in      */,
5        const int    n        /* in      */,
6        float*       trap_p   /* in/out  */) {
7     int my_i = blockDim.x * blockIdx.x + threadIdx.x;
8
9     /* f(x_0) and f(x_n) were computed on the host.  So */
10    /* compute f(x_1), f(x_2), ..., f(x_(n-1))          */
11    if (0 < my_i && my_i < n) {
12       float my_x = a + my_i*h;
13       float my_trap = f(my_x);
14       atomicAdd(trap_p, my_trap);
15    }
16 }  /* Dev_trap */
17
18 /* Host code */
19 void Trap_wrapper(
20       const float  a            /* in  */,
21       const float  b            /* in  */,
22       const int    n            /* in  */,
23       float*       trap_p       /* out */,
24       const int    blk_ct       /* in  */,
25       const int    th_per_blk   /* in  */) {
26
27    /* trap_p storage allocated in main with
28     * cudaMallocManaged */
29    *trap_p = 0.5*(f(a) + f(b));
30    float h = (b-a)/n;
31
32    Dev_trap<<<blk_ct, th_per_blk>>>(a, b, h, n, trap_p);
33    cudaDeviceSynchronize();
34
35    *trap_p = h*(*trap_p);
36 }  /* Trap_wrapper */
```

Program 6.12: CUDA kernel and wrapper implementing trapezoidal rule.

a *pointer* to the memory location to the kernel. That is, we can do something like this:

```
/* Host code */
float* trap_p;
cudaMallocManaged(&trap_p, sizeof(float));
...
```

```
*trap_p = 0.5*(f(a) + f(b));

/* Call kernel */
...

/* After return from kernel */
*trap_p = h*(*trap_p);
```

When we do this, each thread will get its own copy of `trap_p`, but all of the copies of `trap_p` will refer to the same memory location. So `*trap_p` will be shared.

Note that using a pointer instead of a simple float also solves the problem of returning the value of `trap` in Item 4.

A wrapper function

If you look at the code in Program 6.12, you'll see that we've placed most of the code we use before and after calling the kernel in a **wrapper function**, `Trap_wrapper`. A wrapper function is a function whose main purpose is to call another function. It can perform any preparation needed for the call. It can also perform any additional work needed after the call.

6.10.4 Using the correct threads

We assume that the number of threads, `blk_ct*th_per_blk`, is at least as large as the number of trapezoids. Since the serial **for** loop iterates from 1 up to `n`-1, thread 0 and any thread with `my_i` $> n - 1$, shouldn't execute the code in the body of the serial **for** loop. So we should include a test before the main part of the kernel code

```
if (0 < my_i && my_i < n) {
   /* Compute x, f(x), and add in to *trap_p */
   ...
}
```

See Line 11 in Program 6.12.

6.10.5 Updating the return value and `atomicAdd`

This leaves the problem of updating `*trap_p` (Item 3 in the list above). Since the memory location is shared, an update such as

```
*trap_p += my_trap;
```

forms a race condition, and the actual value ultimately stored in `*trap_p` will be unpredictable. We're solving this problem by using a special CUDA library function, `atomicAdd`, to carry out the addition.

An operation carried out by a thread is **atomic** if it appears to all the other threads as if it were "indivisible." So if another thread tries to access the result of the operation or an operand used in the operation, the access will occur either before the operation

started or after the operation completed. Effectively, then, the operation appears to consist of a single, indivisible, machine instruction.

As we saw earlier (see Section 2.4.3), addition is not ordinarily an atomic operation: it consists of several machine instructions. So if one thread is executing an addition, it's possible for another thread to access the operands and the result while the addition is in progress. Because of this, the CUDA library defines several atomic addition functions. The one we're using has the following syntax:

```
__device__ float atomicAdd(
      float*  float_p   /* in/out */,
      float   val       /* in     */);
```

This atomically adds the contents of `val` to the contents of the memory referred to by `float_p` and stores the result in the memory referred to by `float_p`. It returns the value of the memory referred to by `float_p` at the beginning of the call. See Line 14 of Program 6.12.

6.10.6 Performance of the CUDA trapezoidal rule

We can find the run-time of our trapezoidal rule by finding the execution time of the `Trap_wrapper` function. The execution of this function includes all of the computations carried out by the serial trapezoidal rule, including the initialization of `*trap_p` (Line 29) and `h` (Line 30), and the final update to `*trap_p` (Line 35). It also includes all of the calculations in the body of the serial **for** loop in the `Dev_trap` kernel. So we can effectively determine the run-time of the CUDA trapezoidal rule by timing a host function, and we only need to insert calls to our timing functions before and after the call to `Trap_wrapper`. We use the `GET_TIME` macro defined in the `timer.h` header file on the book's website:

```
double start, finish;
...
GET_TIME(start);
Trap_wrapper(a, b, n, trap_p, blk_ct, th_per_blk);
GET_TIME(finish);
printf("Elapsed time for cuda = %e seconds\n",
      finish-start);
```

The same approach can be used to time the serial trapezoidal rule:

```
GET_TIME(start)
trap = Serial_trap(a, b, n);
GET_TIME(finish);
printf("Elapsed time for cpu = %e seconds\n",
      finish-start);
```

Recall from the section on taking timings (Section 2.6.4) that we take a number of timings, and we ordinarily report the minimum elapsed time. However, if the vast majority of the times are much greater (e.g., 1% or 0.1% greater), then the minimum time may not be reproducible. So other users who run the program may get a time

Table 6.5 Mean run-times for serial and CUDA trapezoidal rule (times are in ms).

System	ARM Cortex-A15	Nvidia GK20A	Intel Core i7	Nvidia GeForce GTX Titan X
Clock	2.3 GHz	852 MHz	3.5 GHz	1.08 GHz
SMs, SPs		1, 192		24, 3072
Run-time	33.6	20.7	4.48	3.08

much larger than ours. When this happens, we report the mean or median of the elapsed times.

Now when we ran this program on our hardware, there *were* a number of times that were within 1% of the minimum time. However, we'll be comparing the run-times of this program with programs that had very few run-times within 1% of the minimum. So for our discussion of implementing the trapezoidal rule using CUDA (Sections 6.10–6.13), we'll use the *mean* run-time, and the means are taken over at least 50 executions.

When we run the serial trapezoidal and the CUDA trapezoidal rule functions many times and take the means of the elapsed times, we get the results shown in Table 6.5. These were taken using $n = 2^{20} = 1{,}048{,}576$ trapezoids with $f(x) = x^2 + 1$, $a = -3$, and $b = 3$. The GPUs use 1024 blocks with 1024 threads per block for a total of 1,048,576 threads. The 192 SPs of the GK20A are clearly much faster than a fairly slow conventional processor, an ARM Cortex-A15, but a single core of an Intel Core i7 is much faster than the GK20A. The 3072 SPs on a Titan X *were* 45% faster than the single core of the Intel, but it would seem that with 3072 SPs, we should be able to do better.

6.11 CUDA trapezoidal rule II: improving performance

If you've read the Pthreads or OpenMP chapter, you can probably make a good guess at how to make the CUDA program run faster. For a thread's call to `atomicAdd` to actually be atomic, no other thread can update `*trap_p` while the call is in progress. In other words, the updates to `*trap_p` can't take place simultaneously, and our program may not be very parallel at this point.

One way to improve the performance is to carry out a tree-structured global sum that's similar to the tree-structured global sum we introduced in the MPI chapter (Section 3.4.1). However, because of the differences between the GPU architecture and the distributed-memory CPU architecture, the details are somewhat different.

6.11.1 Tree-structured communication

We can visualize the execution of the "global sum" we implemented in the CUDA trapezoidal rule as a more or less random, linear ordering of the threads. For ex-

Table 6.6 Basic global sum with eight threads.

Time	Thread	my_trap	*trap_p
Start	—	—	9
t_0	5	11	20
t_1	2	5	25
t_2	3	7	32
t_3	7	15	47
t_4	4	9	56
t_5	6	13	69
t_6	0	1	70
t_7	1	3	73

ample, suppose we have only 8 threads and one thread block. Then our threads are $0, 1, \ldots, 7$, and one of the threads will be the first to succeed with the call to `atomicAdd`. Say it's thread 5. Then another thread will succeed. Say it's thread 2. Continuing in this fashion we get a sequence of `atomicAdds`, one per thread. Table 6.6 shows how this might proceed over time. Here, we're trying to keep the computations simple: we're assuming that $f(x) = 2x + 1$, $a = 0$, and $b = 8$. So $h = (8 - 0)/8 = 1$, and the value referenced by `trap_p` at the start of the global sum is

$$0.5 \times (f(a) + f(b)) = 0.5 \times (1 + 17) = 9.$$

What's important is that this approach may serialize the threads. So the computation may require a *sequence* of 8 calculations. Fig. 6.3 illustrates a possible computation.

So rather than have each thread wait for its turn to do an addition into `*trap_p`, we can pair up the threads so that half of the "active" threads add their partial sum to their partner's partial sum. This gives us a structure that resembles a tree (or, perhaps better, a shrub). See Fig. 6.4.

In our figures, we've gone from requiring a sequence of 8 consecutive additions to a sequence of 4. More generally, if we double the number of threads and values (e.g., increase from 8 to 16), we'll double the length of the sequence of additions using the basic approach, while we'll only add one using the second, tree-structured approach. For example, if we increase the number of threads and values from 8 to 16, the first approach requires a sequence of 16 additions, but the tree-structured approach only requires 5. In fact, if there are t threads and t values, the first approach requires a sequence of t additions, while the tree-structured approach requires $\lceil \log_2(t) \rceil + 1$. For example, if we have 1000 threads and values, we'll go from 1000 communications and sums using the basic approach to 11 using the tree-structured approach, and if we have 1,000,000, we'll go from 1,000,000 to 21!

There are two standard implementations of a tree-structured sum in CUDA. One implementation uses shared memory, and in devices with compute capability < 3 this

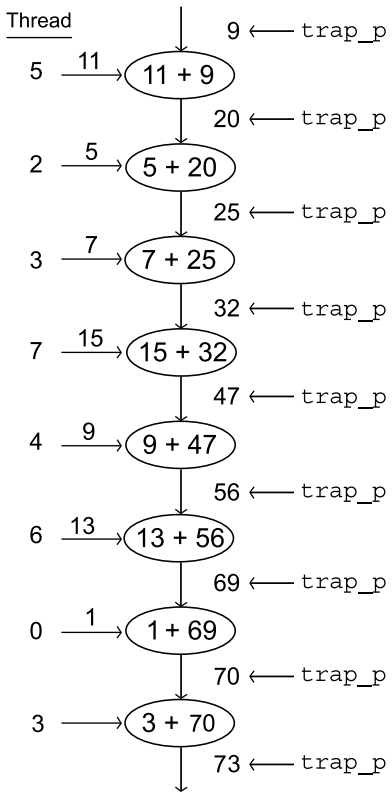

FIGURE 6.3

Basic sum.

is the best implementation. However, in devices with compute capability ≥ 3 there are several functions called **warp shuffles**, that allow a collection of threads within a *warp* to read variables stored by other threads in the warp.

6.11.2 Local variables, registers, shared and global memory

Before we explain the details of how warp shuffles work, let's digress for a moment and talk about memory in CUDA. In Section 6.2 we mentioned that SMs in an Nvidia processor have access to two collections of memory locations: each SM has access to its own "shared" memory, which is accessible only to the SPs belonging to the SM. More precisely, the shared memory allocated for a thread block is only accessible to the threads in that block. On the other hand, all of the SPs and all of the threads have access to "global" memory. The number of shared memory locations is relatively small, but they are quite fast, while the number of global memory locations is relatively large, but they are relatively slow. So we can think of the GPU memory

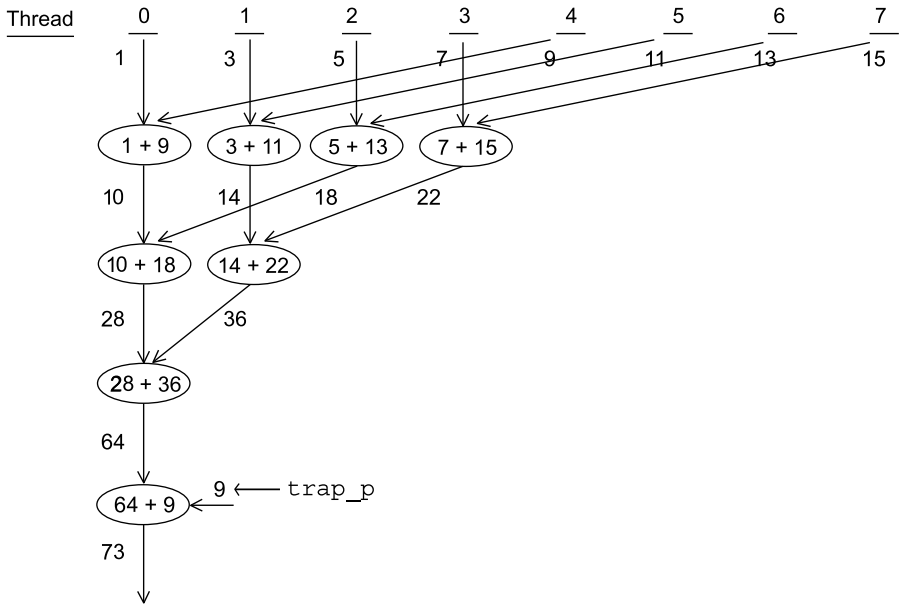

FIGURE 6.4

Tree-structured sum.

Table 6.7 Memory statistics for some Nvidia GPUs.

GPU	Compute Capability	Registers: Bytes per Thread	Shared Mem: Bytes per Block	Global Mem: Bytes per GPU
Quadro 600	2.1	504	48K	1G
GK20A (Jetson TK1)	3.2	504	48K	2G
GeForce GTX Titan X	5.2	504	48K	12G

as a hierarchy with three "levels." At the bottom, is the slowest, largest level: global memory. In the middle is a faster, smaller level: shared memory. At the top is the fastest, smallest level: the registers. For example, Table 6.7 gives some information on relative sizes. Access times also increase dramatically. It takes on the order of 1 cycle to copy a 4-byte int from one register to another. Depending on the system it can take up to an order of magnitude more time to copy from one shared memory location to another, and it can take from two to three orders of magnitude more time to copy from one global memory location to another.

An obvious question here: what about local variables? How much storage is available for them? And how fast is it? This depends on total available memory and program memory usage. If there is enough storage, local variables are stored in registers. However, if there isn't enough register storage, local variables are "spilled" to a

region of global memory that's thread private, i.e., only the thread that owns the local variables can access them.

So as long as we have sufficient register storage, we expect the performance of a kernel to improve if we increase our use of registers and reduce our use of shared and/or global memory. The catch, of course, is that the storage available in registers is tiny compared to the storage available in shared and global memory.

6.11.3 Warps and warp shuffles

In particular, if we can implement a global sum in registers, we expect its performance to be superior to an implementation that uses shared or global memory, and the **warp shuffle** functions introduced in CUDA 3.0 allow us to do this.

In CUDA a **warp** is a set of threads with consecutive ranks belonging to a thread block. The number of threads in a warp is currently 32, although Nvidia has stated that this could change. There is a variable initialized by the system that stores the size of a warp:

```
int warpSize
```

The threads in a warp operate in SIMD fashion. So threads in different warps can execute different statements with no penalty, while threads within the same warp must execute the same statement. When the threads within a warp attempt to execute different statements—e.g., they take different branches in an **if**–**else** statement—the threads are said to have **diverged**. When divergent threads finish executing different statements, and start executing the same statement, they are said to have **converged**.

The rank of a thread within a warp is called the thread's **lane**, and it can be computed using the formula

```
lane = threadIdx.x % warpSize;
```

The warp shuffle functions allow the threads in a warp to read from registers used by another thread in the same warp. Let's take a look at the one we'll use to implement a tree-structured sum of the values stored by the threads in a warp[10]:

```
__device__ float __shfl_down_sync(
          unsigned   mask              /* in */,
          float      var               /* in */,
          unsigned   diff              /* in */,
          int        width = warpSize  /* in */);
```

The `mask` argument indicates which threads are participating in the call. A bit, representing the thread's lane, must be set for each participating thread to ensure that all of the threads in the call have converged—i.e., arrived at the call—before any thread begins executing the call to `__shfl_down_sync`. We'll ordinarily use all the threads in the warp. So we'll usually define

[10] Note that the syntax of the warp shuffles was changed in CUDA 9.0. So you may run across CUDA programs that use the older syntax.

```
mask = 0xffffffff;
```

Recall that 0x denotes a hexadecimal (base 16) value and 0xf is 15_10, which is 1111_2.[11] So this value of mask is 32 1's in binary, and it indicates that every thread in the warp participates in the call to __shfl_down_sync. If the thread with lane l calls __shfl_down_sync, then the value stored in var on the thread with

$$\text{lane} = l + \texttt{diff}$$

is returned on thread l. Since diff has type **unsigned**, it is ≥ 0. So the value that's returned is from a *higher*-ranked thread. Hence the name "shuffle *down*."

We'll only use width = warpSize, and since its default value is warpSize, we'll omit it from our calls.

There are several possible issues:

- What happens if thread l calls __shfl_down_sync but thread $l +$ diff doesn't? In this case, the value returned by the call on thread l is *undefined*.
- What happens if thread l calls __shfl_down_sync but $l +$ diff $\geq$ warpSize? In this case the call will return the value in var already stored on thread l.
- What happens if thread l calls __shfl_down_sync, and $l +$ diff $<$ warpSize, but $l +$ diff $>$ largest lane in the warp. In other words, because the thread block size is not a multiple of warpSize, the last warp in the block has fewer than warpSize threads. Say there are w threads in the last warp, where $0 < w <$ warpSize. Then if

  $$l + \texttt{diff} \geq w,$$

  the value returned by the call is also undefined.

So to avoid undefined results, it's best if

- All the threads in the warp call __shfl_down_sync, and
- All the warps have warpSize threads, or, equivalently, the thread block size (blockDim.x) is a multiple of warpSize.

6.11.4 Implementing tree-structured global sum with a warp shuffle

So we can implement a tree-structured global sum using the following code:

```
__device__ float Warp_sum(float var) {
   unsigned mask = 0xffffffff;

   for (int diff = warpSize/2; diff > 0; diff = diff/2)
      var += __shfl_down_sync_sync(mask, var, diff);
   return var;
}  /* Warp_sum */
```

[11] The subscripts indicate the base.

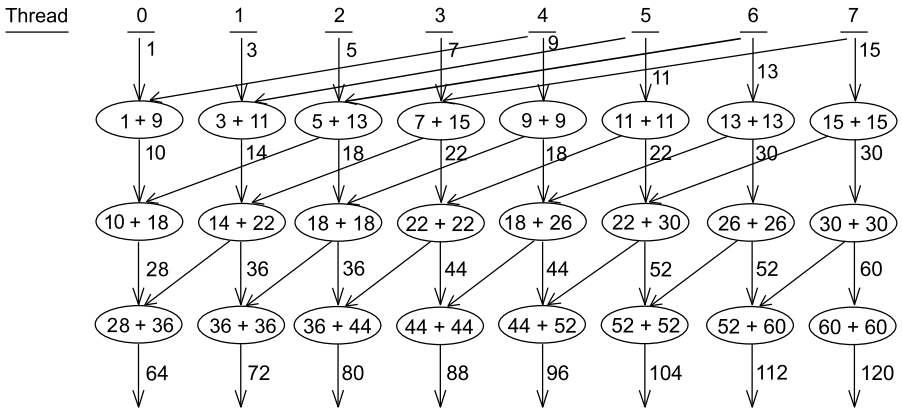

FIGURE 6.5

Tree-structured sum using warp shuffle.

Fig. 6.5 shows how the function would operate if `warpSize` were 8. (The diagram would be illegible if we used a `warpSize` of 32.) Perhaps the most confusing point in the behavior of `__shfl_down_sync` is that when the lane ID

$$l + \texttt{diff} \geq \texttt{warpSize},$$

the call returns the value in the *caller's* `var`. In the diagram this is shown by having only one arrow entering the oval with the sum, and it's labeled with the value just calculated by the thread carrying out the sum. In the row corresponding to `diff` = 4 (the first row of sums), the threads with lane IDs $l = 4, 5, 6$, and 7 all have $l + 4 \geq 8$. So the call to `__shfl_down_sync` returns their current `var` values, 9, 11, 13, and 15, respectively, and these values are doubled, because the return value of the call is added into the calling thread's variable `var`. Similar behavior occurs in the row corresponding to the sums for `diff` = 2 and lane IDs $l = 6$ and 7, and in the last row when `diff` = 1 for the thread with lane ID $l = 7$.

From a practical standpoint, it's important to remember that this implementation will only return the correct sum on the thread with lane ID 0. If all of the threads need the result, we can use an alternative warp shuffle function, `__shfl_xor`. See Exercise 6.6.

6.11.5 Shared memory and an alternative to the warp shuffle

If your GPU has compute capability < 3.0, you won't be able to use the warp shuffle functions in your code, and a thread won't be able to directly access the registers of other threads. However, your code *can* use shared memory, and threads in the same thread block can all access the same shared memory locations. In fact, although shared memory access is slower than register access, we'll see that the shared memory implementation can be just as fast as the warp shuffle implementation.

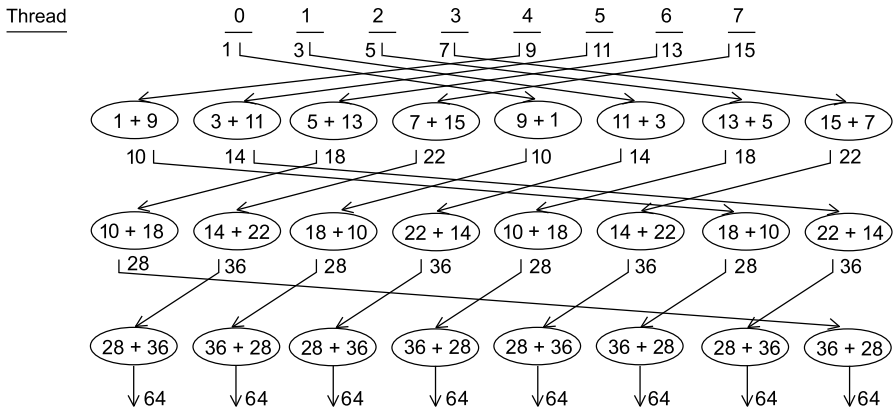

FIGURE 6.6

Dissemination sum using shared memory.

Since the threads belonging to a single warp operate synchronously, we can implement something very similar to a warp shuffle using shared memory instead of registers.

```
__device__ float Shared_mem_sum(float shared_vals[]) {
   int my_lane = threadIdx.x % warpSize;

   for (int diff = warpSize/2; diff > 0;  diff = diff/2) {
      /* Make sure 0 <= source < warpSize */
      int source = (my_lane + diff) % warpSize;
      shared_vals[my_lane] += shared_vals[source];
   }
   return shared_vals[my_lane];
}
```

This should be called by all the threads in a warp, and the array `shared_vals` should be stored in the shared memory of the SM that's running the warp. Since the threads in the warp are operating in SIMD fashion, they effectively execute the code of the function in lockstep. So there's no race condition in the updates to `shared_vals`: all the threads read the values in `shared_vals[source]` before any thread updates `shared_vals[my_lane]`.

Technically speaking, this isn't a tree-structured sum. It's sometimes called a **dissemination sum** or **dissemination reduction**. Fig. 6.6 illustrates the copying and additions that take place. Unlike the earlier figures, this figure doesn't show the direct contributions that a thread makes to its sums: including these lines would have made the figure too difficult to read. Also note that every thread reads a value from another thread in each pass through the **for** statement. After all these values have been added in, every thread has the correct sum—not just thread 0. Although we won't need this for the trapezoidal rule, this can be useful in other applications. Fur-

Table 6.8 Mean run-times for trapezoidal rule using block size of 32 threads (times in ms).

System	ARM Cortex-A15	Nvidia GK20A	Intel Core i7	Nvidia GeForce GTX Titan X
Clock	2.3 GHz	852 MHz	3.5 GHz	1.08 GHz
SMs, SPs		1, 192		24, 3072
Original	33.6	20.7	4.48	3.08
Warp Shuffle		14.4		0.210
Shared Memory		15.0		0.206

We see that on both systems and with both sum implementations, the new programs do significantly better than the original. For the GK20A, the warp shuffle version runs in about 70% of the time of the original, and the shared memory version runs in about 72% of the time of the original. For the Titan X, the improvements are much more impressive: both versions run in less than 7% of the time of the original. Perhaps most striking is the fact that on the Titan X, the warp shuffle is, on average, slightly slower than the shared memory version.

6.13 CUDA trapezoidal rule III: blocks with more than one warp

Limiting ourselves to thread blocks with only 32 threads reduces the power and flexibility of our CUDA programs. For example, devices with compute capability ≥ 2.0 can have blocks with as many as 1024 threads or 32 warps, and CUDA provides a fast barrier that can be used to synchronize *all* the threads in a block. So if we limited ourselves to only 32 threads in a block, we wouldn't be using one of the most useful features of CUDA: the ability to efficiently synchronize large numbers of threads.

So what would a "block" sum look like if we allowed ourselves to use blocks with up to 1024 threads? We could use one of our existing warp sums to add the values computed by the threads in each warp. Then we would have as many as $1024/32 = 32$ warp sums, and we could use one warp in the thread block to add the warp sums.

Since two threads belong to the same warp if their ranks in the block have the same quotient when divided by `warpSize`, to add the warp sums, we can use warp 0, the threads with ranks 0, 1, . . . , 31 in the block.

6.13.1 `__syncthreads`

We might try to use the following pseudocode for finding the sum of the values computed by all the threads in a block:

```
Each thread computes its contribution;
Each warp adds its threads' contributions;
Warp 0 in block adds warp sums;
```

However, there's a race condition. Do you see it? When warp 0 tries to compute the total of the warp sums in the block, it doesn't know whether all the warps in the block have completed their sums. For example, suppose we have two warps, warp 0 and warp 1, each of which has 32 threads. Recall that the threads in a warp operate in SIMD fashion: no thread in the warp proceeds to a new instruction until all the threads in the warp have completed (or skipped) the current instruction. But the threads in warp 0 can operate independently of the threads in warp 1. So if warp 0 finishes computing its sum before warp 1 computes its sum, warp 0 could try to add warp 1's sum to its sum before warp 1 has finished, and, in this case, the block sum could be incorrect.

So we must make sure that warp 0 doesn't start adding up the warp sums until all of the warps in the block are done. We can do this by using CUDA's fast barrier:

```
__device__ void __syncthreads(void);
```

This will cause the threads in the thread block to wait in the call until all of the threads have started the call. Using `__syncthreads`, we can modify our pseudocode so that the race condition is avoided:

```
Each thread computes its contribution;
Each warp adds its threads' contributions;
__syncthreads();
Warp 0 in block adds warp sums;
```

Now warp 0 won't be able to add the warp sums until every warp in the block has completed its sum.

There are a couple of important caveats when we use `__syncthreads`. First, it's critical that *all* of the threads in the block execute the call. For example, if the block contains at least two threads, and our code includes something like this:

```
int my_x = threadIdx.x;
if (my_x < blockDim.x/2)
   __syncthreads();
my_x++;
```

then only half the threads in the block will call `__syncthreads`, and these threads can't proceed until *all* the threads in the block have called `__syncthreads`. So they will wait forever for the other threads to call `__syncthreads`.

The second caveat is that `__syncthreads` only synchronizes the threads in a block. If a grid contains at least two blocks, and if all the threads in the grid call `__syncthreads` then the threads in different blocks will continue to operate independently of each other. So we can't synchronize the threads in a general grid with `__syncthreads`.[12]

[12] CUDA 9 includes an API that allows programs to define barriers across more general collections of threads than thread blocks, but defining a barrier across multiple thread blocks requires hardware support that's not available in processors with compute capability < 6.

6.13.2 More shared memory

If we try to implement the pseudocode in CUDA, we'll see that there's an important detail that the pseudocode doesn't show: after the call to `__syncthreads`, how does warp 0 obtain access to the sums computed by the other warps? It can't use a warp shuffle and registers: the warp shuffles only allow a thread to read a register belonging to another thread when that thread belongs to the same warp, and, for the final warp sum we would like the threads in warp 0 to read registers belonging to threads in *other* warps.

You may have guessed that the solution is to use shared memory. If we use warp shuffles to compute the warp sums, we can just declare a shared array that can store up to 32 floats, and the thread with lane 0 in warp w can store its warp sum in element w of the array:

```
__shared__ float warp_sum_arr[WARPSZ];
int my_warp = threadIdx.x / warpSize;
int my_lane = threadIdx.x % warpSize;
// Threads calculate their contributions;
...
float my_result = Warp_sum(my_trap);
if (my_lane == 0) warp_sum_arr[my_warp] = my_result;
__syncthreads();
// Warp 0 adds the sums in warp_sum_arr
...
```

6.13.3 Shared memory warp sums

If we're using shared memory instead of warp shuffles to compute the warp sums, we'll need enough shared memory for each warp in a thread block. Since shared variables are shared by *all* the threads in a thread block, we need an array large enough to hold the contributions of all of the threads to the sum. So we can declare an array with 1024 elements—the largest possible block size—and partition it among the warps:

```
// Make max thread block size available at compile time
#define MAX_BLKSZ 1024
...
___shared__ float thread_calcs[MAX_BLKSZ];
```

Now each warp will store its threads' calculations in a subarray of `thread_calcs`:

```
float* shared_vals = thread_calcs + my_warp*warpSize;
```

In this setting a thread stores its contribution in the subarray referred to by `shared_vals`:

```
shared_vals[my_lane] = f(my_x);
```

Now each warp can compute the sum of its threads' contributions by using our shared memory implementation that uses blocks with 32 threads:

```
float my_result = Shared_mem_sum(shared_vals);
```

To continue we need to store the warp sums in locations that can be accessed by the threads in warp 0 in the block, and it might be tempting to try to make a subarray of `thread_calcs` do "double duty." For example, we might try to use the first 32 elements for both the contributions of the threads in warp 0, and the warp sums computed by the warps in the block. So if we have a block with 32 warps of 32 threads, warp w might store its sum in `thread_calcs[w]` for $w = 0, 1, 2, \ldots, 31$.

The problem with this approach is that we'll get another race condition. When can the other warps safely overwrite the elements in warp 0's block? After a warp has completed its call to `Shared_mem_sum`, it would need to wait until warp 0 has finished its call to `Shared_mem_sum` before writing to `thread_calcs`:

```
float my_result = Shared_mem_sum(shared_vals);
__syncthreads();
if (my_lane == 0) thread_calcs[my_warp] = my_result.
```

This is all well and good, but warp 0 still can't proceed with the final call to `Shared_mem_sum`: it must wait until all the warps have written to `thread_calcs`. So we would need a *second* call to `__syncthreads` before warp 0 could proceed:

```
if (my_lane == 0) thread_calcs[my_warp] = my_result.
__syncthreads();
// It's safe for warp 0 to proceed ...
if (my_warp == 0)
   my_result = Shared_mem_sum(thread_calcs);
```

Calls to `__syncthreads` are fast, but they're not free: every thread in the thread block will have to wait until all the threads in the block have called `__syncthreads`. So this can be costly. For example, if there are more threads in the block than there are SPs in an SM, the threads in the block won't all be able to execute simultaneously. So some threads will be delayed reaching the second call to `__syncthreads`, and all of the threads in the block will be delayed until the last thread is able to call `__syncthreads`. So we should only call `__syncthreads()` when we have to.

Alternatively, each warp could store its warp sum in the "first" element of its subarray:

```
float my_result = Shared_mem_sum(shared_vals);
if (my_lane == 0) shared_vals[0] = my_result;
__syncthreads();
...
```

It might at first appear that this would result in a race condition when the thread with lane 0 attempts to update `shared_vals`, but the update is OK. Can you explain why?

6.13.4 Shared memory banks

However, this implementation may not be as fast as possible. The reason has to do with details of the design of shared memory: Nvidia divides the shared memory on

Table 6.9 Shared memory banks: Columns are memory banks. The entries in the body of the table show subscripts of elements of `thread_calcs`.

	Bank					
	0	**1**	**2**	**...**	**30**	**31**
Subscripts	0	1	2	...	30	31
	32	33	34	...	62	63
	64	65	66	...	94	95
	96	97	98	...	126	127
	⋮	⋮	⋮	⋱	⋮	⋮
	992	993	994	...	1022	1023

an SM into 32 "banks" (16 for GPUs with compute capability < 2.0). This is done so that the 32 threads in a warp can simultaneously access shared memory: the threads in a warp can simultaneously access shared memory when each thread accesses a different bank.

Table 6.9 illustrates the organization of `thread_calcs`. In the table, the columns are banks, and the rows show the subscripts of consecutive elements of `thread_calcs`. So the 32 threads in a warp can simultaneously access the 32 elements in any one of the rows, or, more generally, if each thread access is to a different column.

When two or more threads access different elements in a single bank (or column in the table), then those accesses must be serialized. So the problem with our approach to saving the warp sums in elements 0, 32, 64, ..., 992 is that these are all in the same bank. So when we try to execute them, the GPU will serialize access, e.g., element 0 will be written, then element 32, then element 64, etc. So the writes will take something like 32 times as long as it would if the 32 elements were stored in different banks, e.g., a row of the table.

The details of bank access are a little complicated and some of the details depend on the compute capability, but the main points are

- If each thread in a warp accesses a different bank, the accesses can happen simultaneously.
- If multiple threads access different memory locations in a single bank, the accesses must be serialized.
- If multiple threads read the same memory location in a bank, the value read is broadcast to the reading threads, and the reads are simultaneous.

The CUDA programming Guide [11] provides full details.

Thus we could exploit the use of the shared memory banks if we stored the results in a contiguous subarray of shared memory. Since each thread block can use at least 16 Kbytes of shared memory, and our "current" definition of `shared_vals` only uses at most 1024 floats or 4 Kbytes of shared memory, there is plenty of shared memory available for storing 32 more floats.

So if we're using shared memory warp sums, a simple solution is to declare *two* arrays of shared memory: one for storing the computations made by each thread, and another for storing the warp sums.

```
__shared__ float thread_calcs[MAX_BLKSZ];
__shared__ float warp_sum_arr[WARPSZ];
float* shared_vals = thread_calcs + my_warp*warpSize;
...
float my_result = Shared_mem_sum(shared_vals);
if (my_lane == 0) warp_sum_arr[my_warp] = my_result;
__syncthreads();
...
```

6.13.5 Finishing up

The remaining codes for the warp sum kernel and the shared memory sum kernel are very similar. First warp 0 computes the sum of the elements in `warp_sum_arr`. Then thread 0 in the block adds the block sum into the total across all the threads in the grid using `atomicAdd`. Here's the code for the shared memory sum:

```
if (my_warp == 0) {
   if (threadIdx.x >= blockDim.x/warpSize)
      warp_sum_arr[threadIdx.x] = 0.0;
   blk_result = Shared_mem_sum(warp_sum_arr);
}

if (threadIdx.x == 0) atomicAdd(trap_p, blk_result);
```

In the test `threadIdx.x > blockDim.x/warpSize` we're checking to see if there are fewer than 32 warps in the block. If there are, then the final elements in `warp_sum_arr` won't have been initialized. For example, if there are 256 warps in the block, then

```
blockDim.x/warpSize = 256/32 = 8
```

So there are only 8 warps in a block and we'll have only initialized elements $0, 1, \ldots, 7$ of `warp_sum_arr`. But the warp sum function expects 32 values. So for the threads with `threadIdx.x` >= 8, we assign

```
warp_sum_arr[threadIdx.x] = 0.0;
```

For the sake of completeness, Program 6.15 shows the kernel that uses shared memory. The main differences between this kernel and the kernel that uses warp shuffles are that the declaration of the first shared array isn't needed in the warp shuffle version, and, of course, the warp shuffle version calls `Warp_sum` instead of `Shared_mem_sum`.

6.13.6 Performance

Before moving on, let's take a final look at the run-times for our various versions of the trapezoidal rule. (See Table 6.10.) The problem is the same: find the area under

```
1  __global__ void Dev_trap(
2        const float  a         /* in  */,
3        const float  b         /* in  */,
4        const float  h         /* in  */,
5        const int    n         /* in  */,
6        float*       trap_p    /* out */) {
7     __shared__ float thread_calcs[MAX_BLKSZ];
8     __shared__ float warp_sum_arr[WARPSZ];
9     int my_i = blockDim.x * blockIdx.x + threadIdx.x;
10    int my_warp = threadIdx.x / warpSize;
11    int my_lane = threadIdx.x % warpSize;
12    float* shared_vals = thread_calcs + my_warp*warpSize;
13    float blk_result = 0.0;
14
15    shared_vals[my_lane] = 0.0f;
16    if (0 < my_i && my_i < n) {
17       float my_x = a + my_i*h;
18       shared_vals[my_lane] = f(my_x);
19    }
20
21    float my_result = Shared_mem_sum(shared_vals);
22    if (my_lane == 0) warp_sum_arr[my_warp] = my_result;
23    __syncthreads();
24
25    if (my_warp == 0) {
26       if (threadIdx.x >= blockDim.x/warpSize)
27          warp_sum_arr[threadIdx.x] = 0.0;
28       blk_result = Shared_mem_sum(warp_sum_arr);
29    }
30
31    if (threadIdx.x == 0) atomicAdd(trap_p, blk_result);
32 }  /* Dev_trap */
```

Program 6.15: CUDA kernel implementing trapezoidal rule and using shared memory. This version can use large thread blocks.

the graph of $y = x^2 + 1$ between $x = -3$ and $x = 3$ using $2^{20} = 1{,}048{,}576$ trapezoids. However, instead of using a block size of 32 threads, this version uses a block size of 1024 threads. The larger blocks provide a significant advantage on the GK20A: the warp shuffle version is more than 10% faster than the version that uses 32 threads per block, and the shared memory version is about 5% faster. On the Titan X, the performance improvement is huge: the warp shuffle version is more than 30% faster, and the shared memory version is more than 25% faster. So on the faster GPU, reducing the number of threads calling `atomicAdd` was well worth the additional programming effort.

Table 6.10 Mean run-times for trapezoidal rule using arbitrary block size (times in ms).

System	ARM Cortex-A15	Nvidia GK20A	Intel Core i7	Nvidia GeForce GTX Titan X
Clock	2.3 GHz	852 MHz	3.5 GHz	1.08 GHz
SMs, SPs		1, 192		24, 3072
Original	33.6	20.7	4.48	3.08
Warp Shuffle, 32 ths/blk		14.4		0.210
Shared Memory, 32 ths/blk		15.0		0.206
Warp Shuffle		12.8		0.141
Shared Memory		14.3		0.150

6.14 Bitonic sort

Bitonic sort is a somewhat unusual sorting algorithm, but it has the virtue that it can be parallelized fairly easily. Even better, it can be parallelized so that the threads operate independently of each other for clearly defined segments of code. On the down side, it's not a very intuitive sorting algorithm, and our implementation will require frequent use of barriers. In spite of this, our parallelization with CUDA will be considerably faster than a single core implementation on a CPU.

The algorithm proceeds by building subsequences of keys that form *bitonic* sequences. A bitonic sequence is a sequence that first increases and then decreases.[13] The butterfly structure (see Section 3.4.4) is at the heart of bitonic sort. However, now, instead of defining the structure by communicating between pairs of processes, we define it by the use of compare-swap operations.

6.14.1 Serial bitonic sort

To see how this works suppose that n is a positive integer that is a power of 2, and we have a list of n integer keys. Any pair of consecutive elements can be turned into either an increasing or a decreasing sequence by a compare-swap: these are two-element butterflies. If we have a four-element list, we can first create a bitonic sequence with a couple of compare swaps or two-element butterflies. We then use a four-element butterfly to create a sorted list.

As an example, suppose our input list is $\{40, 20, 10, 30\}$. Then the first two elements can be turned into an increasing sequence by a compare-swap:

```
if (list[0] > list[1]) Swap(&list[0], &list[1]);
```

[13] Technically, a bitonic sequence is either a sequence that first increases and then decreases, or it is a sequence that can be converted to such a sequence by one or more circular shifts. For example, 3, 5, 4, 2, 1 is a bitonic sequence, since it increases and then decreases, but 5, 4, 2, 1, 3 is also a bitonic sequence, since it can be converted to the first sequence by a circular shift.