

DEPARTMENT OF COMPUTER SCIENCE AND DESIGN

Notes of Lesson

FACULTY NAME : DR. PAVITHRA A C

SUBJECT TITLE AND CODE : PARALLEL COMPUTING – BCS702

SEMESTER AND SCHEME : VII SEMESTER



INSTITUTIONAL MISSION AND VISION

Objectives

- To provide quality education and groom top-notch professionals, entrepreneurs and leaders for different fields of engineering, technology and management.
- To open a Training-R & D-Design-Consultancy cell in each department, gradually introduce doctoral and postdoctoral programs, encourage basic & applied research in areas of social relevance, and develop the institute as a center of excellence.
- To develop academic, professional and financial alliances with the industry as well as the academia at national and transnational levels.
- To develop academic, professional and financial alliances with the industry as well as the academia at national and transnational levels.
- To cultivate strong community relationships and involve the students and the staff in local community service.
- To constantly enhance the value of the educational inputs with the participation of students, faculty, parents and industry.

MODULE – 1

INTRODUCTION TO PARALLEL PROGRAMMING: PARALLEL HARDWARE AND PARALLEL SOFTWARE

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

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

PARALLEL HARDWARE

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

1. SIMD systems

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

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

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

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

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

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

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

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

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

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

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

Vector processors

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

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

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

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

Thus, vector operations are SIMD.

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

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

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

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

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

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

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

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

Graphics processing units

Real-time graphics application programming interfaces, or APIs, use points, lines, and triangles to internally represent the surface of an object. They use a **graphics 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 ALUs (e.g., 80) on each GPU processing core.

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

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

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

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

2. MIMD systems

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

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

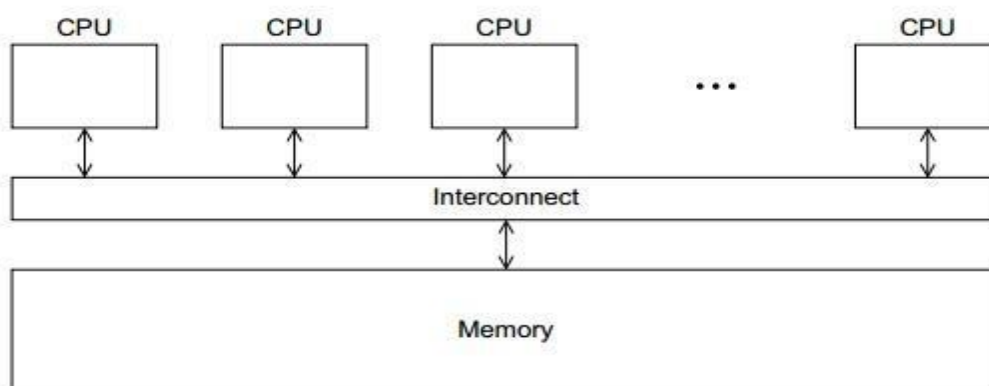
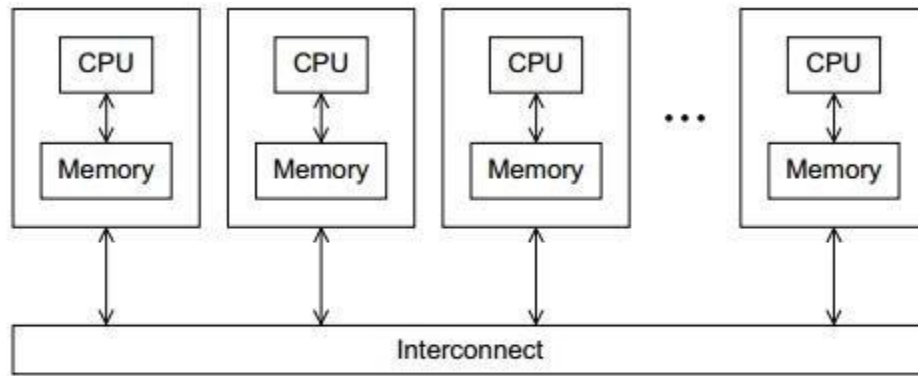


FIGURE 2.3

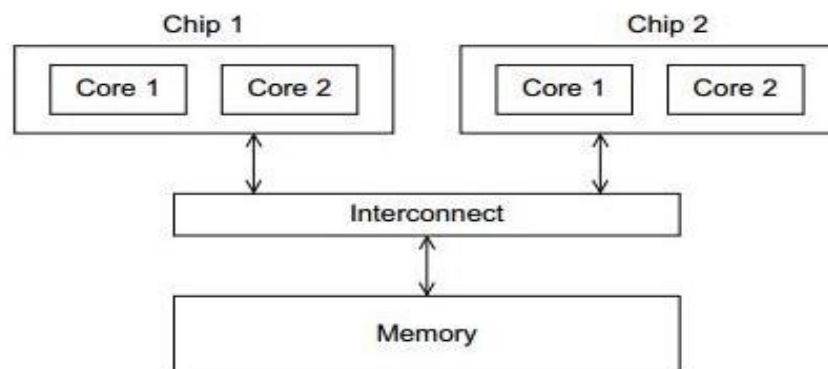
A shared-memory system

**FIGURE 2.4**

A distributed-memory system

Shared-memory systems

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

**FIGURE 2.5**

A UMA multicore system

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

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

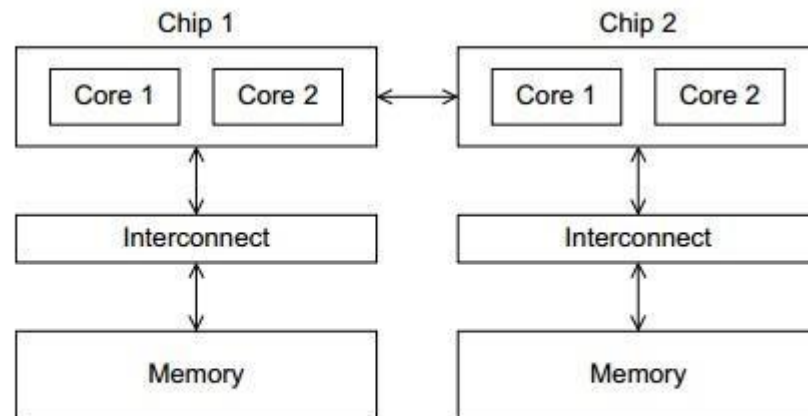


FIGURE 2.6

A NUMA multicore system

Distributed-memory systems

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

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

3. Interconnection networks

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

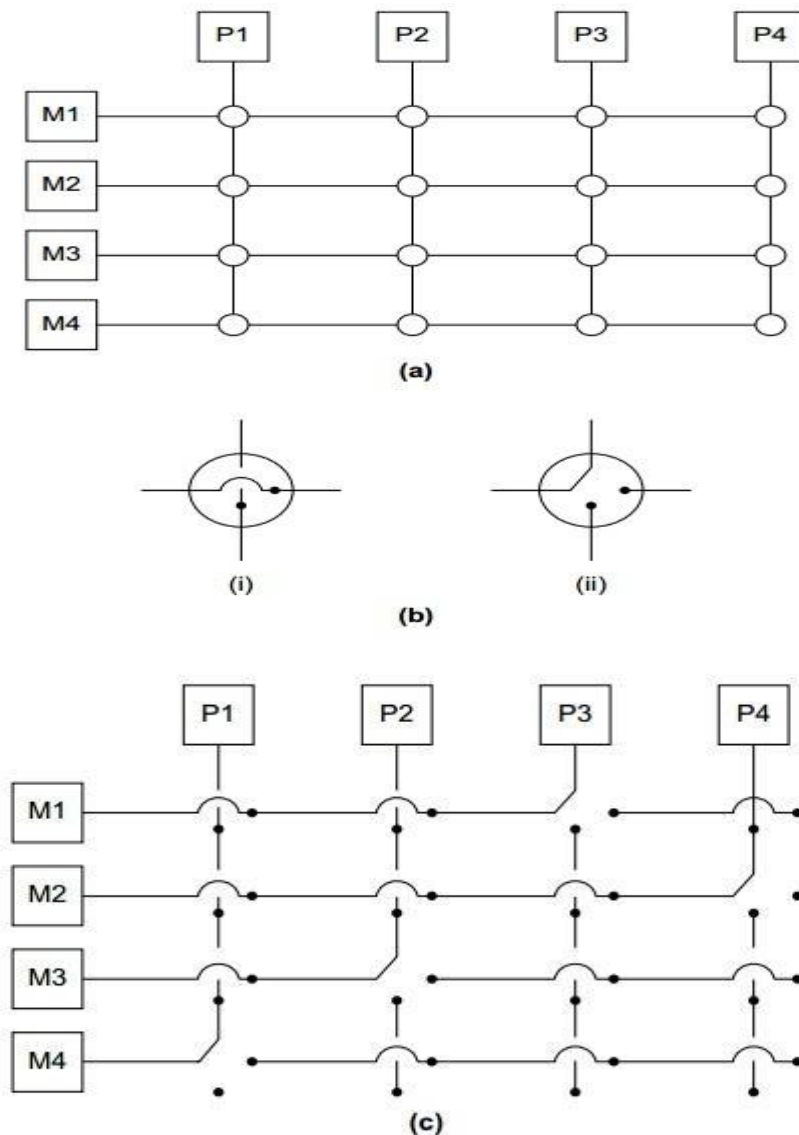
Although some of the interconnects have a great deal in common, there are enough differences to make it worthwhile to treat interconnects for shared-memory and distributed-memory separately.

Shared-memory interconnects

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

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

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

**FIGURE 2.7**

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

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

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

Distributed-memory interconnects

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

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

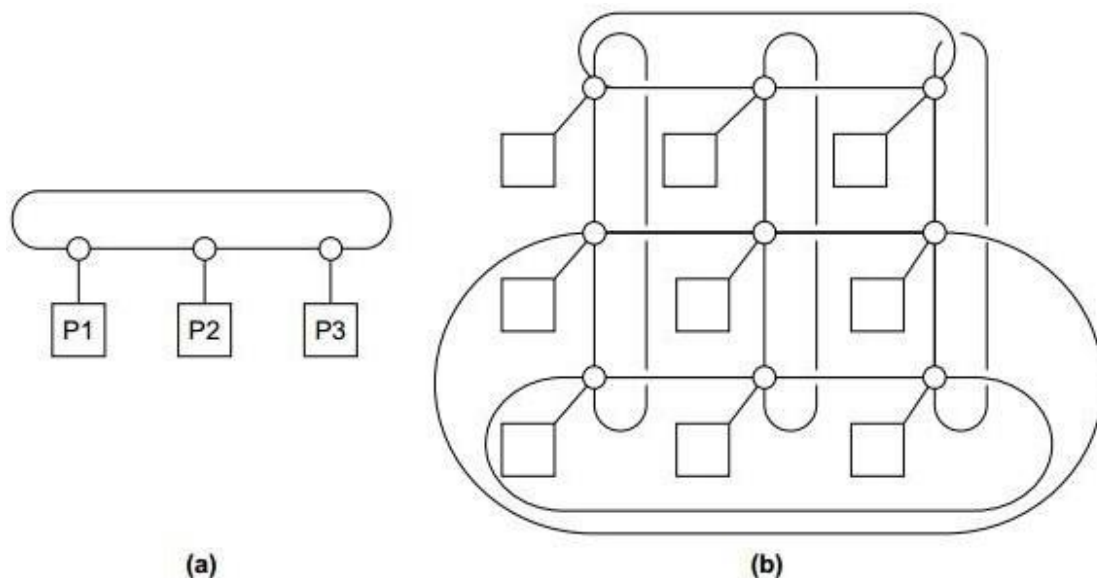
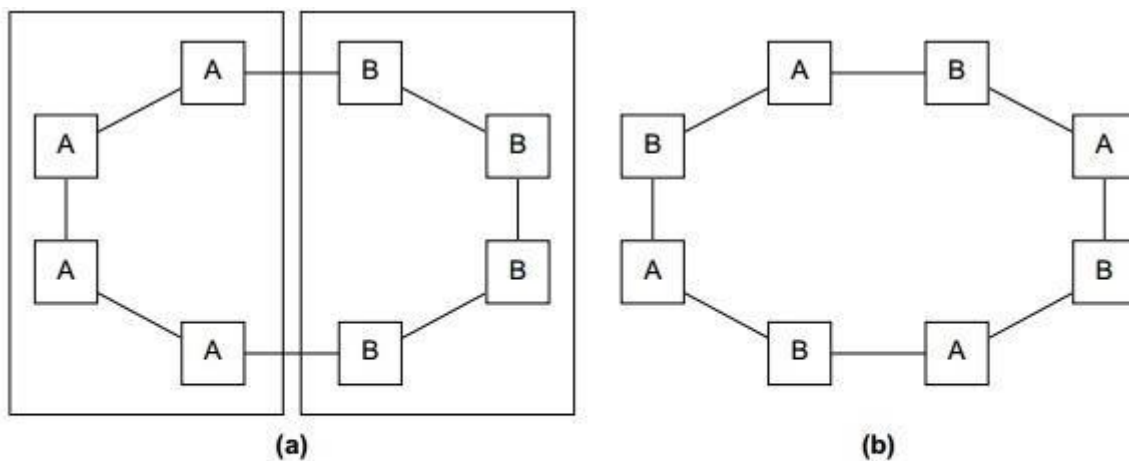


FIGURE 2.8

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

**FIGURE 2.9**

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

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

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

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

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

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

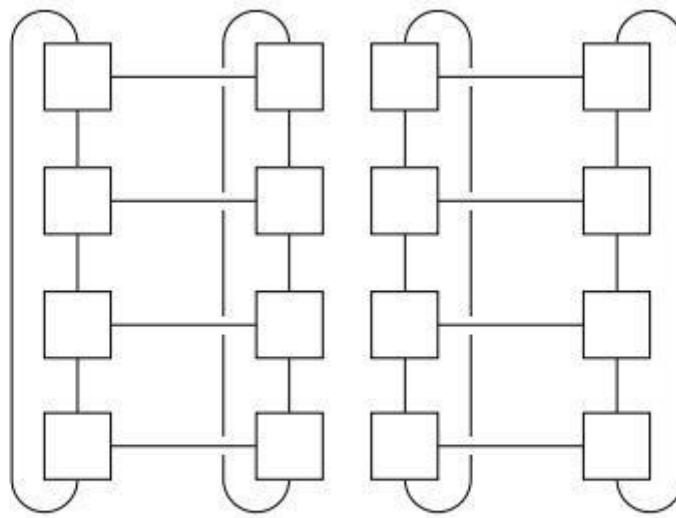


FIGURE 2.10

A bisection of a toroidal mesh

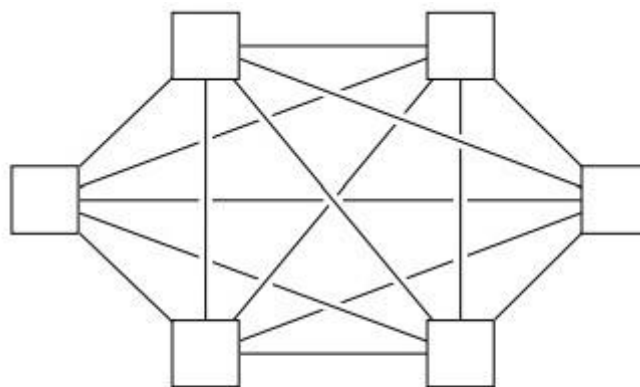
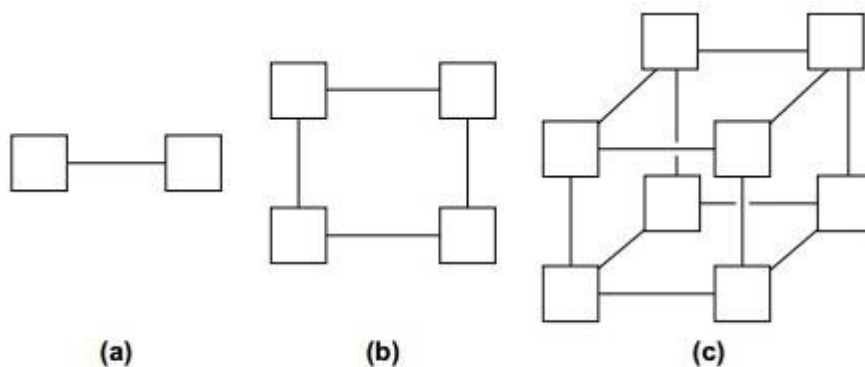


FIGURE 2.11

A fully connected network

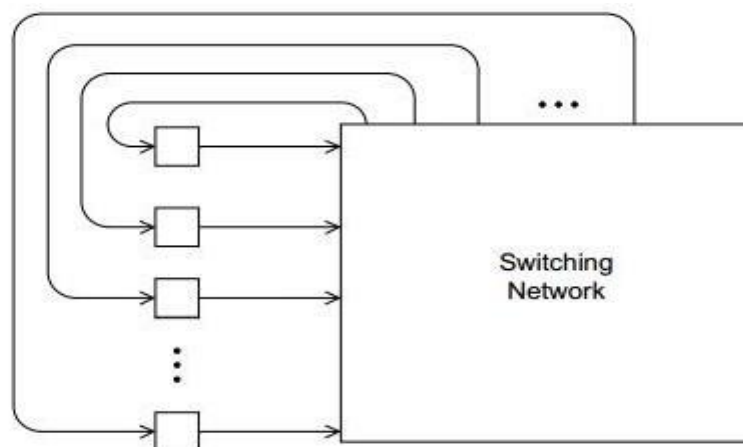
The **hypercube** is a highly connected direct interconnect that has been used in actual systems. Hypercubes are built inductively: A one-dimensional hypercube is a fully-connected system

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

**FIGURE 2.12**

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

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

**FIGURE 2.13**

A generic indirect network

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

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

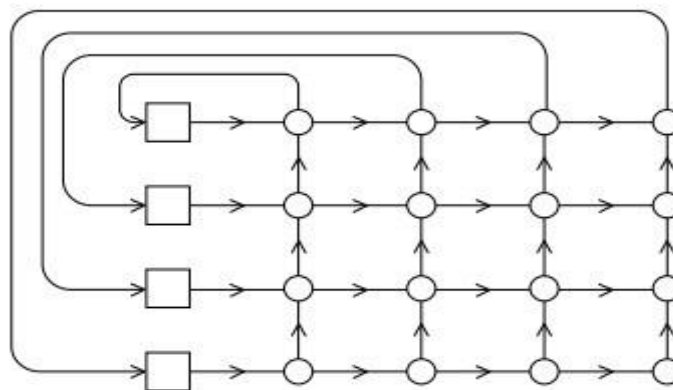


FIGURE 2.14

A crossbar interconnect for distributed-memory

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

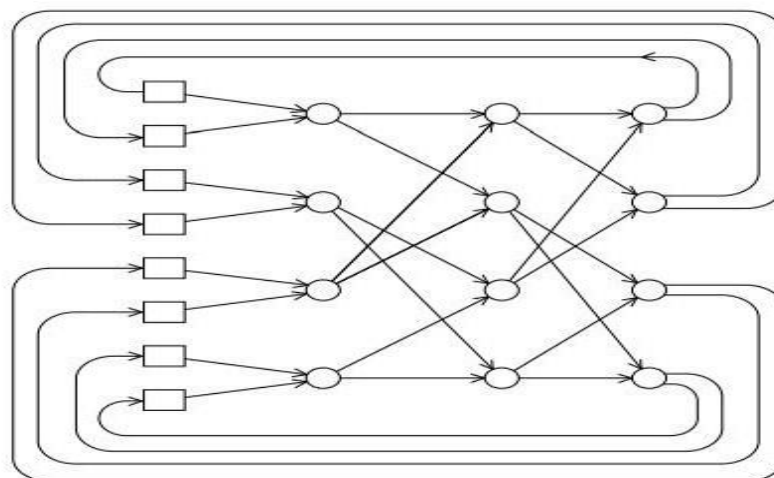
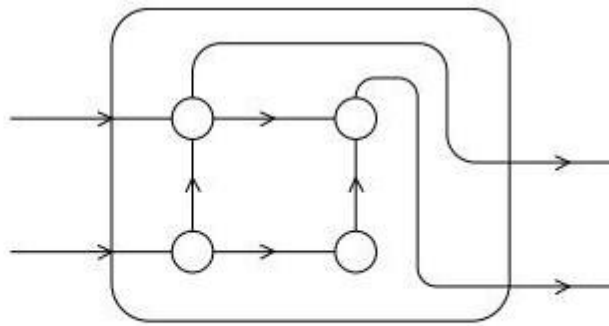


FIGURE 2.15

An omega network

**FIGURE 2.16**

A switch in an omega network

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

Latency and bandwidth

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

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

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

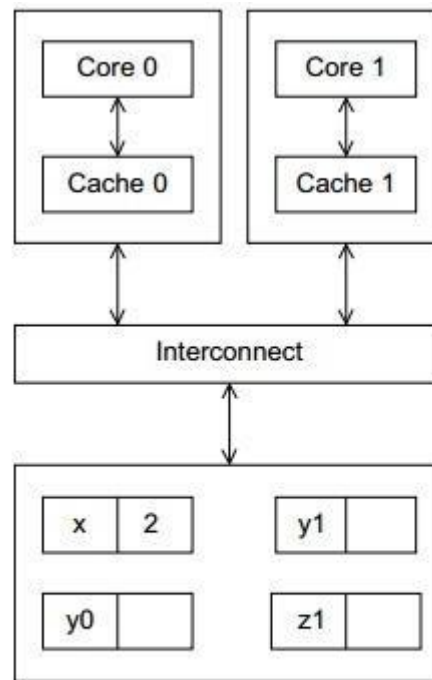
the time required for any fixed overhead involved in transmitting data. For example, if we're sending a message between two nodes in a distributed-memory system, a message is not just raw data. It might include the data to be transmitted, a destination address, some information specifying the size of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

4. Cache coherence

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

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

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

**FIGURE 2.17**

A shared-memory system with two cores and two caches

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

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

Snooping cache coherence

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

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

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

Directory-based cache coherence

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

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

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

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

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

False sharing

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

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

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

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

```
/* Private variables */  
int i, j, iter count;  
  
/ Shared variables initialized by one core */  
int m, n, core count  
  
double y[m];  
  
iter count = m/core count  
  
/* Core 0 does this */
```

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

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

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

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

```
...
```

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

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

5. Shared-memory versus distributed-memory

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

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

Coordinating the processes/threads

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

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

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

So for this example, the programmer only needs to

Divide the work among the processes/threads

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

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

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

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

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

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

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

Arrange for the processes/threads to synchronize.

Arrange for communication among the processes/threads.

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

Shared-memory

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

Dynamic and static threads

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

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

Nondeterminism

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

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

Then the output could be

Thread 0 > my_val = 7

Thread 1 > my_val = 19

but it could also be

Thread 1 > my_val = 19

Thread 0 > my_val = 7

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

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

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

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

```
x += my_val;
```

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

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

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

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

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

```
my_val = Compute_val(my rank); Lock(&add my_val lock);
x += my_val;
```

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

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

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

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

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

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

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

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

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

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

Thread safety

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

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

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

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

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

Distributed-memory

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

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

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

Message-passing

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

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

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

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

Receive, process 0 prints the message.

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

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

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

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

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

One-sided communication

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

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

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

Partitioned global address space languages

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

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

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

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

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

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

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

MODULE – 2

GPU PROGRAMMING, PROGRAMMING HYBRID SYSTEMS, MIMD SYSTEMS, GPUS, PERFORMANCE

INPUT AND OUTPUT

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, [35]. 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, we, as developers, would like 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, 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 is a little less obvious. Should the input be divided among the processes? Or should only a single process be allowed to call `scanf`? The vast majority of systems allow at least one process to call `scanf`—usually process 0—while some allow more processes. Once again, there are some systems that don't allow any processes to call `scanf`.

When multiple processes *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 may be broken up by the output of another process. For input, the data read by each process may be different on each run, even if the same input is used.

In order 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.

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?

1. Speedup and efficiency

Usually the best we can hope to do is to equally divide the work 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. If we call the serial run-time T_{serial} and our parallel run-time T_{parallel} , then the best we can hope for is $T_{\text{parallel}} = T_{\text{serial}}/p$. When this happens,

we say that our parallel program has linear speedup.

In practice, we're unlikely to get 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 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 very 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, that is, more threads will probably mean more threads need to access a critical section. 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 $S = p$, which is unusual. Furthermore, as p increases, we 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.

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

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

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

Table 2.5 Speedups and Efficiencies of a 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

It's clear 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 Figure 2.18, and the efficiencies are plotted in Figure 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. This behavior is quite common. Many parallel programs are developed by 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

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

Furthermore, as the problem size is increased, T_{overhead} often grows more slowly than T_{serial} . When this is the case the speedup and the efficiency will increase. See Exercise 2.16. This is

what your intuition should tell you: there's more work for the processes/threads to do, so the relative amount of time spent coordinating the work of the processes/threads should be less.

A final issue to consider is what values of T_{serial} should be used when report-ing speedups and efficiencies. Some authors say that T_{serial} should be the run-time of the fastest program on the fastest processor available. 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 par-allel 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.

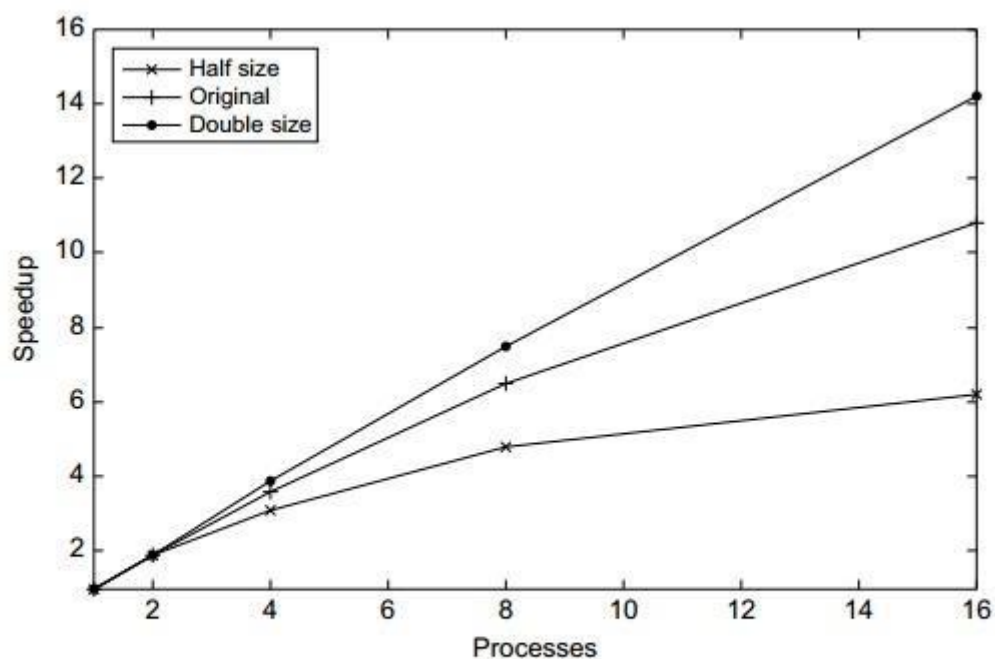
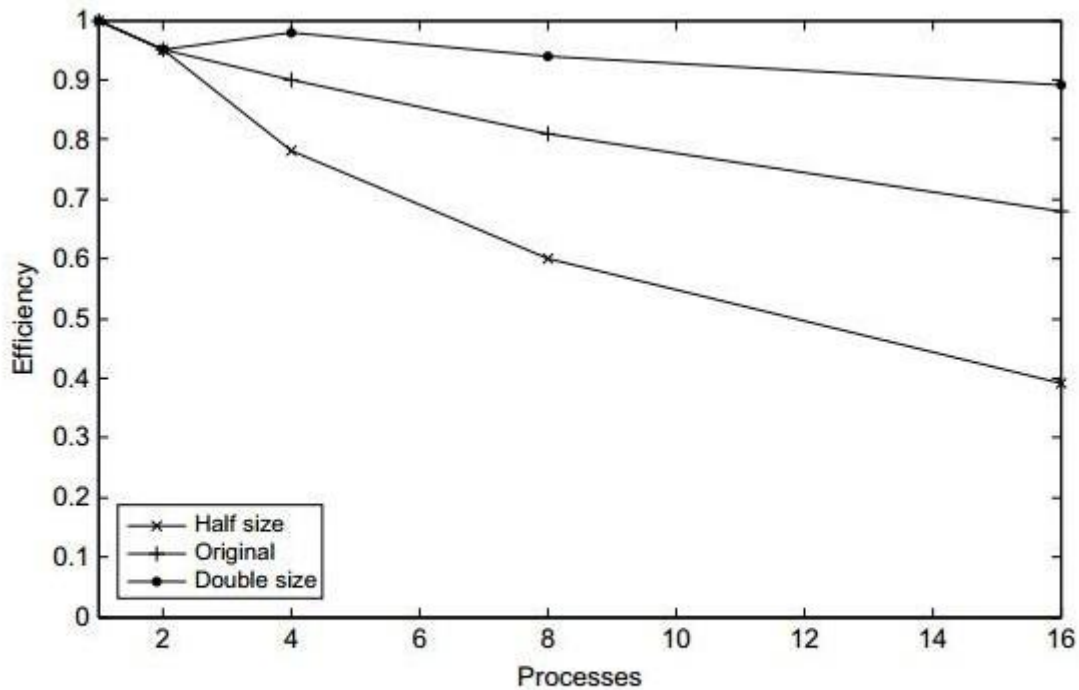


FIGURE 2.18

Speedups of parallel program on different problem sizes

**FIGURE 2.19**

Efficiencies of parallel program on different problem sizes

2. Amdahl's law

Back in the 1960s, Gene Amdahl made an observation [2] 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. Further 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 $0.1 T_{\text{serial}} = 2$. The fraction S must therefore be smaller than

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

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.

3. Scalability

The word “scalable” has a wide variety of informal uses. Indeed, we've used it several times already. Roughly speaking, a technology is scalable if it can handle ever-increasing problem sizes. However, in discussions of 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 microsec-onds, 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 and 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.

4. Taking timings

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 in order 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 2.22 for a brief discussion of some issues 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. 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);  
shared double 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 process-es/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 process-es/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.

MODULE-3

Distributed memory programming with MPI –

MPI functions, The trapezoidal rule in MPI, Dealing with I/O, Collective communication, MPI-derived datatypes, Performance evaluation of MPI programs, A parallel sorting algorithm.

Distributed memory programming with MPI

In the world of **parallel MIMD (Multiple Instruction, Multiple Data)** systems, we typically distinguish between **distributed-memory** and **shared-memory** architectures. In a **distributed-memory system** (see Fig. 3.1), each **core** has access to its **own private memory**, and cores are connected via a **network**. A core can **only access its own memory** directly. Communication with other cores requires **explicit message-passing**.

In contrast, a **shared-memory system** (see Fig. 3.2) allows all cores to access a **common global memory**. This shared access simplifies some aspects of programming but introduces new challenges like **synchronization** and **cache coherence**.

To program distributed-memory systems, we use **message-passing programming models**. A program running on each core-memory pair is referred to as a **process**, and communication occurs using **send** and **receive** functions. The widely-used model is **MPI (Message-Passing Interface)**—a **standardized library** of functions callable from **C** and **Fortran**.

MPI supports not only basic point-to-point communication but also **collective communication**, where **multiple processes** participate in operations like **broadcast**, **gather**, **scatter**, and **reduce**. These **collective functions** are crucial for implementing scalable distributed algorithms.

While learning MPI, it's also necessary to understand the fundamental concerns in message-passing systems, such as **data partitioning**, which involves splitting the problem and data among different processes, and **distributed I/O**, which deals with efficiently managing input and output across processes.

These programming aspects directly impact **parallel program performance**, making it essential to study **communication cost**, **load balancing**, and **synchronization overheads** in distributed-memory environments.

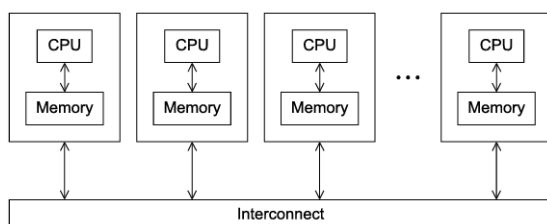


FIGURE 3.1
A distributed memory system.

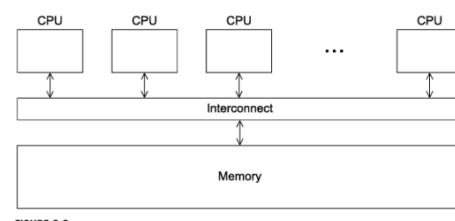


FIGURE 3.2
A shared memory system.

3.1 Getting Started

The traditional "hello, world" program from **Kernighan and Ritchie's text** introduces basic program output. In parallel programming using **MPI (Message-Passing Interface)**, we adapt

this by letting **only one process (rank 0)** handle the output, while the **other processes send messages** to it.

MPI identifies processes by **non-negative integer ranks**, ranging from 0 to $p-1$, where **p** is the **total number of processes**.

In this **MPI-based hello world** example (see **Program 3.1**), the workflow is:

- **Process 0** acts as the **receiver and printer**.
- All **other processes** use **MPI_Send** to send greeting messages to process 0.
- Process 0 uses **MPI_Recv** to **receive messages** and prints them.

3.1.1 Compilation and Execution

The method for **compiling and executing MPI programs** depends on the system in use. On most systems, developers write the code using a **text editor** and then use the **command line** to compile and run it.

```

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.

Compilation

Most systems provide an *mpicc* command to compile MPI programs. *mpicc* is a **wrapper script** around the C compiler. It automatically includes:

- Necessary **header files**
- Required **MPI libraries**

Example Command:

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

- -g enables debugging information
- -Wall enables all warnings
- -o mpi_hello specifies the output executable

Execution

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!
```

- The command `mpiexec` **starts the specified number of instances** of the compiled MPI program.
- It may also control **which cores** the instances are assigned to.
- The **MPI implementation** ensures that these processes can **communicate with each other** as needed for the program to function.

3.1.2 MPI Programs

MPI programs are written in **standard C**, which means they start just like any other C program. For instance, the program includes the usual **C header files** such as `<stdio.h>` and `<string.h>`, and it defines a standard **main function**.

What distinguishes an MPI program is the inclusion of the `<mpi.h>` header file (as shown in **Line 3** of the code). This header is essential because it provides **function prototypes**, **macro definitions**, **type definitions**, and all other declarations required to compile an **MPI-based program**.

An important convention in MPI programming is the **naming scheme** used for its identifiers. All identifiers provided by MPI begin with **MPI_**. For function names and **MPI-defined types**, the first letter after the underscore is capitalized. For example:

- Function: MPI_Init, MPI_Send, MPI_Comm_rank
- Type: MPI_Comm, MPI_Status

In contrast, **MPI-defined macros and constants** are written in **all capital letters**, such as MPI_CHAR, MPI_COMM_WORLD, and MPI_STATUS_IGNORE. This consistent naming makes it easy to distinguish between **MPI-provided elements** and **user-defined identifiers**.

3.1.3 MPI_Init and MPI_Finalize

In **Line 12** of the program, the call to **MPI_Init** is crucial as it tells the **MPI system** to perform all necessary **initialization steps**. These steps may include allocating **message buffers** and assigning **ranks** to processes. Importantly, **no other MPI function** should be called before MPI_Init. Its syntax is:

```
int MPI_Init(int* argc_p, char*** argv_p);
```

The parameters argc_p and argv_p are pointers to the program's command-line arguments (argc and argv). However, if the program doesn't use command-line arguments, we can simply pass **NULL** for both. Like most MPI functions, MPI_Init returns an **integer error code**, which we usually **ignore** to avoid cluttering the code.

At the end of the program (**Line 31**), the call to **MPI_Finalize** signals that the program is done using MPI and any resources allocated can be released. Its syntax is very simple:

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

As a **rule**, no MPI calls should be made **after** MPI_Finalize.

A typical structure of an **MPI program** includes these functions as:

```
#include <mpi.h>

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);
    // MPI code
    MPI_Finalize();
    return 0;
}
```

3.1.4 Communicators, MPI_Comm_size, and MPI_Comm_rank

In **MPI**, a **communicator** is a group of processes that can communicate with each other. One of the key tasks of MPI_Init is to create the default communicator **MPI_COMM_WORLD**, which includes **all processes** initiated when the program starts. To retrieve details about the processes in this communicator, we use:

```
int MPI_Comm_size(MPI_Comm comm, int* comm_sz_p);
int MPI_Comm_rank(MPI_Comm comm, int* my_rank_p);
```

The parameter `comm` is of type `MPI_Comm` and refers to the communicator (typically `MPI_COMM_WORLD`). The function `MPI_Comm_size` stores the **total number of processes** in `comm_sz_p`, while `MPI_Comm_rank` stores the **rank** (or ID) of the **calling process** in `my_rank_p`. Conventionally, we use the variables `comm_sz` for total number of processes and `my_rank` for the process's rank within `MPI_COMM_WORLD`.

3.1.5 SPMD Programs

Despite the fact that different processes might perform **different tasks**, we usually compile and execute a **single program** in MPI. This programming style is known as **Single Program, Multiple Data (SPMD)**. In our example, although **process 0** prints messages and the other processes **send messages**, all processes still run the **same compiled code**. The **if-else construct** in lines 16–29 helps differentiate the behavior based on each process's **rank**.

This approach makes the program **flexible and scalable**—it can run with **any number of processes**, such as 1, 4, 1000, or even more, depending on the system's resources. Even though MPI doesn't require this, it is **good practice** to write MPI programs that can adapt to **varying numbers of processes**, since the **available resources** can change from system to system or over time.

3.1.6 Communication

In the MPI greeting program, **each process except process 0** constructs a message using `sprintf`, which formats the message into a **string** instead of printing it to the console. Then, the message is sent to **process 0** using `MPI_Send`.

Process 0, on the other hand, prints its own greeting directly using `printf`, and then uses a **loop** to **receive messages** from all other processes (1 to `comm_sz - 1`) using `MPI_Recv`. This pattern illustrates a basic example of **point-to-point communication** in MPI, where **one process sends** and **another receives** a message.

3.1.7 MPI_Send

The function `MPI_Send` is used to **transmit messages** between processes. Its prototype is:

```
int MPI_Send(
    void* msg_buf_p,      // pointer to message buffer
    int msg_size,         // number of elements
    MPI_Datatype msg_type, // datatype of elements (e.g., MPI_CHAR)
    int dest,             // destination rank
    int tag,              // message tag
    MPI_Comm communicator // communicator group
);
```

- `msg_buf_p` points to the **data buffer** (e.g., greeting).
- `msg_size` and `msg_type` define **how much data** is sent (e.g., `strlen(greeting)+1` with type `MPI_CHAR`).
- `dest` is the **rank of the destination** process.
- `tag` is a **message identifier**, used to distinguish between different types of messages.
- `communicator` (e.g., `MPI_COMM_WORLD`) defines the **scope** of communication.

Table 3.1 Some predefined MPI datatypes.

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

Important: MPI types such as `MPI_CHAR`, `MPI_INT`, etc., are of type `MPI_Datatype`, not regular C types.

Example Use-Case for Tags: If process 1 sends some data to process 0 and it includes two kinds of data (e.g., print vs. compute), **tags** like 0 and 1 can help **differentiate** them.

Communicators help isolate communications. For example, if you're using **two independent MPI-based libraries** (say, atmosphere and ocean simulations), using **separate communicators** ensures their messages don't interfere with each other. This avoids complicated **tag-based coordination** and improves **modularity**.

MPI_Recv

The corresponding **receive function** is `MPI_Recv`:

```
int MPI_Recv(
    void* msg_buf_p,      // receive buffer
    int buf_size,         // max elements to receive
    MPI_Datatype buf_type, // type of elements
    int source,           // source rank
    int tag,              // message tag
    MPI_Comm communicator, // communicator group
    MPI_Status* status_p  // status info (can use MPI_STATUS_IGNORE)
);
```

- msg_buf_p is where the received message will be **stored**.
- buf_size and buf_type define **how much can be stored**.
- source specifies the **rank of the sender**.
- tag must **match** the sender's tag.
- communicator must match that of the sender.
- status_p is used to **inspect** additional info (e.g., actual source or tag); it can be ignored using MPI_STATUS_IGNORE.

MPI_Recv needs to **match** the send exactly in **data type, size, communicator, source, and tag** to complete successfully.

3.1.9 Message Matching

In MPI, for a message sent by one process to be **successfully received** by another, several parameters must **match** between the MPI_Send and MPI_Recv calls.

Matching Conditions

Suppose process **q** sends a message:

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

And process **r** receives a message:

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

The **message is received** successfully only if the following conditions hold:

- recv_comm == send_comm
- recv_tag == send_tag
- dest == r
- src == q

These are necessary, but not sufficient. The **buffers** must also be **compatible**:

If `recv_type == send_type` **and** `recv_buf_sz ≥ send_buf_sz`, the message can be received

Unpredictable Order Handling

In many cases, the receiver doesn't know the **order** in which messages will arrive. For example:

- Process 0 distributes work to others (1, 2, ..., `comm_sz-1`)
- Those processes return results to process 0
- Some tasks may finish earlier, but process 0 **must wait in order** if using fixed ranks. This creates delays.

Solution: MPI_ANY_SOURCE

To allow receiving from **any process**, MPI provides the **wildcard constant** `MPI_ANY_SOURCE`.

```
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);
}
```

This allows process 0 to **receive results as they come in**, **not** in process-rank order.

Wildcard for Tags

Similarly, if a process is expecting **multiple message types** (e.g., with different tags), it can use:

`MPI_ANY_TAG`

This enables the process to **receive any incoming message**, regardless of its tag.

Wildcard constants (`MPI_ANY_SOURCE`, `MPI_ANY_TAG`) can only be used in **receive** operations.

- Senders **must specify** both **destination rank** and **tag** explicitly.
- Hence, MPI uses a **push** model (data is sent to receivers), **not pull**. **No**

wildcard for communicators:

- Both sender and receiver **must agree** on the **communicator** (`MPI_COMM_WORLD`, etc.).

3.1.10 The `status_p` Argument

In **MPI**, a process can receive a message without prior knowledge of the **message size**, **sender**, or **tag**. This is made possible through the final argument of `MPI_Recv`, which is a pointer to a structure of type `MPI_Status`. This structure includes fields such as **`MPI_SOURCE`**, **`MPI_TAG`**, and **`MPI_ERROR`**. After a receive call like

```
MPI_Status status;
```

```
MPI_Recv(..., &status);
```

the **sender's rank** and the **message tag** can be accessed using `status.MPI_SOURCE` and `status.MPI_TAG`.

However, to determine the **number of elements** received, we use the function `MPI_Get_count`, since this information isn't directly stored in `MPI_Status`. Its syntax is:

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

This function computes the count based on the **data type** and stores it in `count_p`. The reason for using a function instead of a struct field is that the **count depends on data type** and might involve computations like dividing the number of received bytes by the size of the data type. To avoid **unnecessary overhead**, this calculation is only done when explicitly requested by the programmer.

3.1.11 Semantics of MPI_Send and MPI_Recv

When a process executes `MPI_Send`, the message is assembled with both **data** and **envelope information** such as **destination rank**, **sender rank**, **tag**, **communicator**, and **message size**. After this, two behaviors are possible: the MPI system may either **buffer** the message or **block** the sending process. If **buffered**, the message is stored internally and `MPI_Send` returns immediately. If **blocked**, the function waits until it can transmit the message. Importantly, once `MPI_Send` returns, we cannot be certain that the message has been transmitted—only that the **send buffer is safe for reuse**. For more control, MPI offers **alternative send functions** with different behaviors.

The behavior of `MPI_Send` also depends on the **message size**. Many implementations use a **cutoff size**: messages smaller than the cutoff are typically buffered, while larger ones cause blocking.

Unlike `MPI_Send`, `MPI_Recv` **always blocks** until a matching message has been received. Thus, after it returns, we are guaranteed that a message resides in the **receive buffer** (barring any errors). There also exists a **non-blocking alternative** for receiving, which checks availability and returns regardless.

MPI ensures that messages are **non-overtaking**: if **process q** sends two messages to **process r**, the **first message must be received before** the second. However, messages from **different processes** (e.g., **q** and **t**) may arrive in **any order**, regardless of their sending sequence. This flexibility is necessary since MPI cannot enforce network performance—e.g., **a message from Mars can't be guaranteed to arrive before one from San Francisco**, even if sent earlier.

3.1.12 Some Potential Pitfalls

A key pitfall in MPI programming lies in the semantics of `MPI_Recv`, which **blocks** until a **matching send** is issued. If no such send exists, the process will **hang indefinitely**. This highlights the importance of ensuring that **every receive has a corresponding send**. Coding

errors, such as mismatched **tags** or incorrect **source/destination ranks**, can prevent a match, causing unexpected hangs or, worse, messages being received by the **wrong process**.

On the sending side, a call to **MPI_Send** may either **block** or **buffer**. If it blocks and no matching receive is available, the **sending process will hang**. If it buffers but still lacks a matching receive, the **message will be lost**. Hence, meticulous attention must be paid during both **design and coding** to maintain correct communication patterns and avoid these silent failures.

3.2 The Trapezoidal Rule in MPI

While simple programs like printing messages help us understand MPI basics, the real benefit of **parallel programming** comes when we apply it to more useful computations. A good example is the **trapezoidal rule for numerical integration**, which is a method used to estimate the definite integral of a function. In a **serial implementation**, this involves dividing the integration interval into smaller subintervals, computing the function at each point, and applying the trapezoidal formula. In **MPI**, this process can be **parallelized** by dividing the integration interval among multiple processes. Each process computes the area under the curve over its assigned subinterval and sends its **partial result** to a **designated process** (usually rank 0), which then combines them to produce the **final result**. This example demonstrates how **computation** and **communication** are coordinated in MPI programs, moving from basic I/O tasks to solving actual mathematical problems.

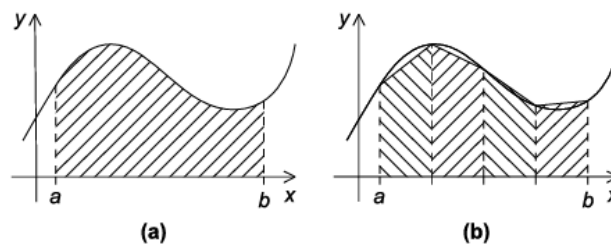


FIGURE 3.3

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

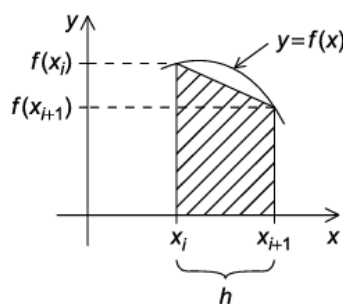


FIGURE 3.4

One trapezoid.

3.2.1 The Trapezoidal Rule

The **trapezoidal rule** provides a method to **numerically approximate definite integrals** by dividing the interval $[a, b]$ into **n equal subintervals** and summing up the areas of **trapezoids** under the curve. As shown in **Figure 3.3**, we consider the area between the graph of $y=f(x)$, two vertical lines, and the x -axis. Each subinterval is treated as a **trapezoid**, with its area approximated using the function values at the subinterval's

endpoints. According to **Figure 3.4**, the height of each trapezoid is the **interval width** $h = \frac{b-a}{n}$, and its area is calculated as:

$$\text{Area} = (h / 2) * [f(x_i) + f(x_{i+1})]$$

Summing the areas of all trapezoids gives the total approximation:

$$\text{Total Area} \approx h * [f(x_0)/2 + f(x_1) + f(x_2) + \dots + f(x_{n-1}) + f(x_n)/2]$$

Here, the x -values are uniformly spaced:

$$x_0 = a, x_1 = a + h, \dots, x_n = b$$



A **serial pseudocode** for implementing this is:

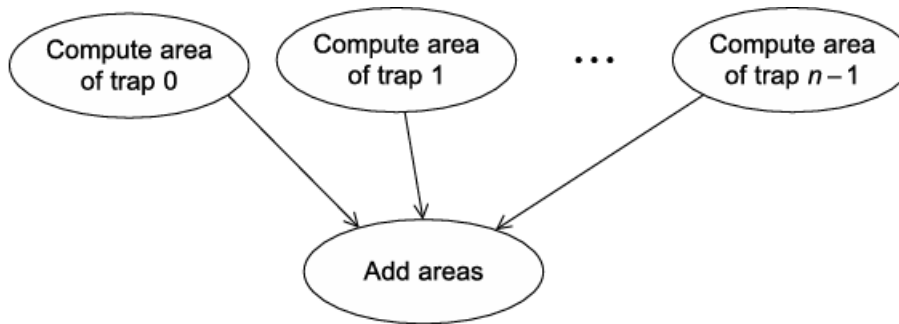
```
// Input: a, b, n
h = (b - a) / n;
approx = (f(a) + f(b)) / 2.0;
for (i = 1; i <= n - 1; i++) {
    x_i = a + i * h;
    approx += f(x_i);
}
approx *= h;
```

This method provides a foundation for **parallelization using MPI**, where the interval and corresponding computations can be distributed across multiple processes.

3.2.2 Parallelizing the Trapezoidal Rule

To **parallelize** the trapezoidal rule, we follow the standard steps in parallel program design:

(1) *partition* the computation into tasks, (2) *identify communication*, (3) *aggregate tasks*, and (4) *map tasks to cores*. In this case, we identify two main tasks: **computing the area** of each trapezoid and **summing the results**. As shown in **Figure 3.5**, each area-computing task sends its result to a central summing task.

**FIGURE 3.5**

Tasks and communications for the trapezoidal rule.

Assuming the number of processes `comm_sz` evenly divides the number of trapezoids `n`, we can assign **$n/\text{comm_sz}$** trapezoids to each process. Each process computes a **local integral** over its subinterval `[local_a, local_b]`. After computing, processes **send their local results** to **process 0**, which **receives** and **sums** them to get the final approximation.

The **pseudocode** is as follows:

```

Get a, b, n;
h = (b - a)/n;
local_n = n / comm_sz;
local_a = a + my_rank * local_n * h;
local_b = local_a + local_n * h;
local_integral = Trap(local_a, local_b, local_n, h);

```

```

if (my_rank != 0)
    Send local_integral to process 0;
else {
    total_integral = local_integral;
    for (proc = 1; proc < comm_sz; proc++) {
        Receive local_integral from proc;
        total_integral += local_integral;
    }
}

```

```

if (my_rank == 0)
    print result;

```

Here, `Trap()` is a function implementing the serial trapezoidal rule. Note the distinction between **local variables** (like `local_a`, `local_b`, and `local_n`, which are private to each process) and **global variables** (`a`, `b`, and `n`, which are meaningful across all processes). Although these definitions differ from traditional programming terms, the context in parallel programming makes them intuitive.

MPI program shown in Program 3.2. The `Trap` function is just an implementation of the serial trapezoidal rule. See Program 3.3.

```

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.

```

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.

3.3 Dealing with I/O

The current version of the **parallel trapezoidal rule** has a major **limitation**: it only works for the interval **[0, 3]** using **1024 trapezoids**. While we *could* manually edit the code and recompile for new values, this is **inconvenient** compared to simply typing in **three new input values** during execution.

Therefore, it's necessary to **address the issue of user input** in

parallel programs. And while discussing **input handling**, it's also a good opportunity to look at **output mechanisms** in parallel MPI programs.

3.3.1 Output

In both the “**greetings**” program and the **trapezoidal rule program**, we assumed that **process 0** can write to **stdout** via `printf`, and this behaves as expected. While the **MPI standard** doesn't strictly define which processes can access I/O devices, most **MPI implementations** allow all processes in **MPI_COMM_WORLD** to access both **stdout** and **stderr**. However, there is **no automatic scheduling** of output access. If **multiple processes** try to write simultaneously to **stdout**, the output order becomes **unpredictable**, and one process's output might even be **interrupted** by another's.

For instance, when running a program (as in *Program 3.4*) where each process prints a line, the output order may appear sequential with **five processes**, but becomes **nondeterministic** with **six or more processes**. This behavior occurs because the processes **compete** for access to **shared output**, leading to **race conditions** in printing. The result is **nondeterminism**—output may differ between runs.

To maintain a **predictable output order**, it's the programmer's responsibility to **manage output**. A common solution is to have each **non-zero process** send its output as a **message** to **process 0**, which then **prints** in **rank order**—as demonstrated in the “**greetings**” program.

```
#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.

3.3.2 Input

Unlike output, **most MPI implementations** allow **only process 0** in **MPI_COMM_WORLD** to access **stdin**. This restriction is **logical**, since if multiple processes accessed **stdin**, it would be ambiguous as to **which process** should receive **which part** of the input. For example, should **process 0** receive the first line and **process 1** the second, or should input be distributed **character-by-character**?

To handle input using functions like **scanf**, MPI programs typically **branch on process rank**. That is, **process 0** reads the input and then **broadcasts** or **sends** it to the other processes. For instance, in the **parallel trapezoidal rule program**, the `Get_input` function (see *Program 3.5*) allows **process 0** to read the values for

```

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.

a, **b**, and **n**, and then **sends** these values to each of the other processes. This communication follows the same pattern as in the “**greetings**” program, except that now, **process 0** sends the data, while the other processes **receive** it.

To use this input function, a call to `Get_input` should be inserted inside **main**, after initializing **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

In our trapezoidal rule MPI program, one inefficiency is evident in how the **global sum** is computed. After calculating local integrals, every process **sends** its value to **process 0**, which

alone performs the summation. This results in **poor workload distribution**, as process 0 does most of the work, while the other processes become idle after sending. Such imbalance is inefficient, especially when resources are underutilized. A better approach would be to **distribute the summation work**, so all processes contribute equally, similar to how students can collaboratively add numbers rather than relying on one to sum them all.

3.4.1 Tree-Structured Communication

An efficient solution is the use of **tree-structured communication**, such as a **binary tree** (see Fig. 3.6). Initially, pairs like processes 1 & 0, 3 & 2, 5 & 4, and 7 & 6 exchange and add values. In successive steps, the **intermediate results** are sent and summed by fewer processes, until **process 0** receives the final total. This method reduces the number of receives and additions **by process 0** from $\text{comm_sz} - 1$ to just $\log_2(\text{comm_sz})$ operations. For example, with $\text{comm_sz} = 1024$, the number of operations drops from **1023** to just **10**, significantly improving performance.

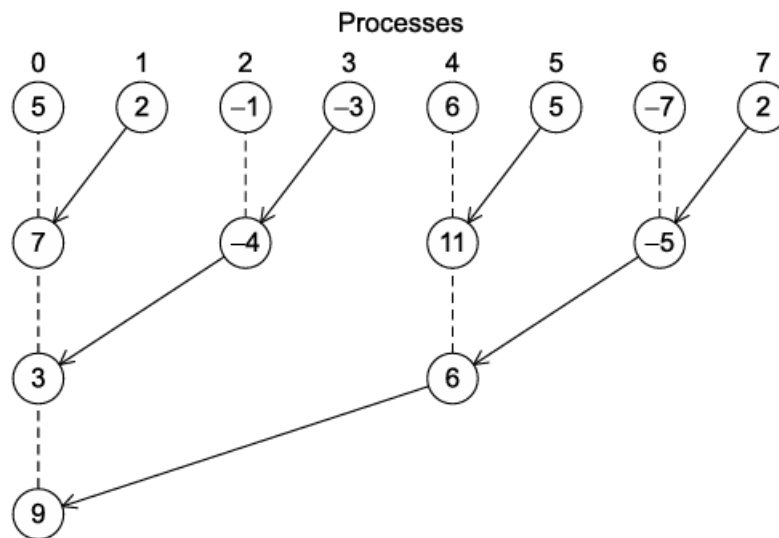
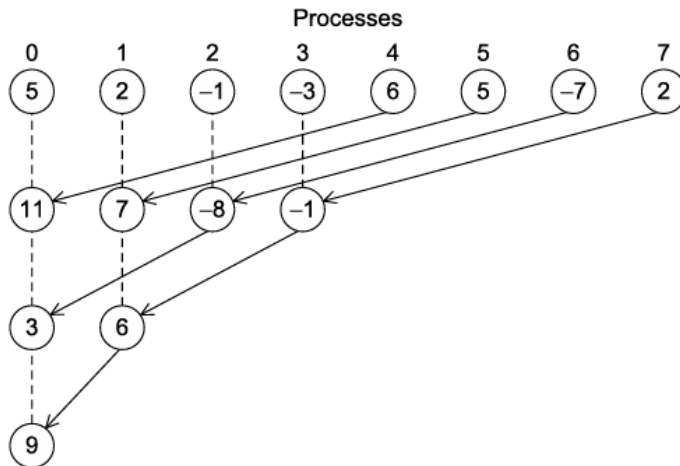


FIGURE 3.6

A tree-structured global sum.

The advantage lies in **concurrent execution**—several adds and receives happen **simultaneously** in early phases, which reduces **total execution time**. However, **coding this manually** can be complex, especially when choosing optimal **pairings** (e.g., pairing 0 & 4, 1 & 5, etc., as in Fig. 3.7). Each strategy may perform differently based on the **system architecture** and **number of processes**, making it difficult to predict the best approach without **experimentation** or **system-specific optimization**.

**FIGURE 3.7**

An alternative tree-structured global sum.

3.4.2 MPI_Reduce

To avoid redundant effort in writing optimized global sum functions, MPI offers a built-in collective communication routine called **MPI_Reduce**, which shifts the responsibility of optimization to the **MPI implementation**. This function enables multiple processes in a communicator (like MPI_COMM_WORLD) to perform a **reduction operation**—such as **sum**, **maximum**, **minimum**, or **product**—on data distributed across processes. Such **collective communications** contrast with **point-to-point operations** (MPI_Send, MPI_Recv) by involving **all processes** in the communicator.

The prototype for MPI_Reduce is:

```
int MPI_Reduce(
    void* input_data_p,    // input buffer
    void* output_data_p,   // result buffer (valid only at root)
    int count,             // number of elements
    MPI_Datatype datatype, // type of data (e.g., MPI_DOUBLE)
    MPI_Op operator,       // operation to perform (e.g., MPI_SUM)
    int dest_process,      // rank of root process
    MPI_Comm comm          // communicator (e.g., MPI_COMM_WORLD)
);
```

The **fifth parameter**—the operator of type MPI_Op—is central to its generality. MPI defines several built-in operations for this parameter, such as **MPI_SUM**, **MPI_PROD**, **MPI_MAX**, and **MPI_MIN**, which are listed in **Table 3.2**. Programmers can also define **custom reduction operations** using MPI_Op_create.

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

In the **parallel trapezoidal rule program**, this call replaces the manual summation logic in the root process:

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

This line aggregates local_int from each process into total_int at **process 0**. For arrays, MPI_Reduce works similarly. For instance, with double local_x[N];, one can compute a component-wise sum as:

```
MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Thus, **MPI_Reduce** provides a simple, efficient, and portable way to perform global reductions across processes.

3.4.3 Collective vs. Point-to-Point Communications

Collective communications in MPI differ significantly from point-to-point communications in the following ways:

1. **All processes in the communicator must call the same collective function.** For instance, calling MPI_Reduce on one process and MPI_Recv on another is incorrect and will likely **cause the program to hang or crash**.
2. **Arguments passed by each process must be compatible.** If one process uses dest_process = 0 and another uses dest_process = 1 in a call to MPI_Reduce, the result is erroneous.
3. **The output_data_p argument is used only by the destination process**, but all processes must still pass a valid argument (e.g., NULL for non-destination processes).
4. **Collective communications do not use tags**, unlike point-to-point operations. They are matched **only by communicator and the order of calls**. As shown in **Table 3.3**, if each process calls MPI_Reduce twice, the sequence determines matching—not variable names—so values accumulate in the order of function invocation.

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, ...)

A final **cautionary point**: using the **same buffer for input and output** (e.g., MPI_Reduce(&x, &x, ...)) is **illegal in MPI**, due to **aliasing** restrictions. This restriction, rooted in Fortran compatibility, ensures that output arguments don't overlap with input memory, preventing undefined or erroneous behavior.

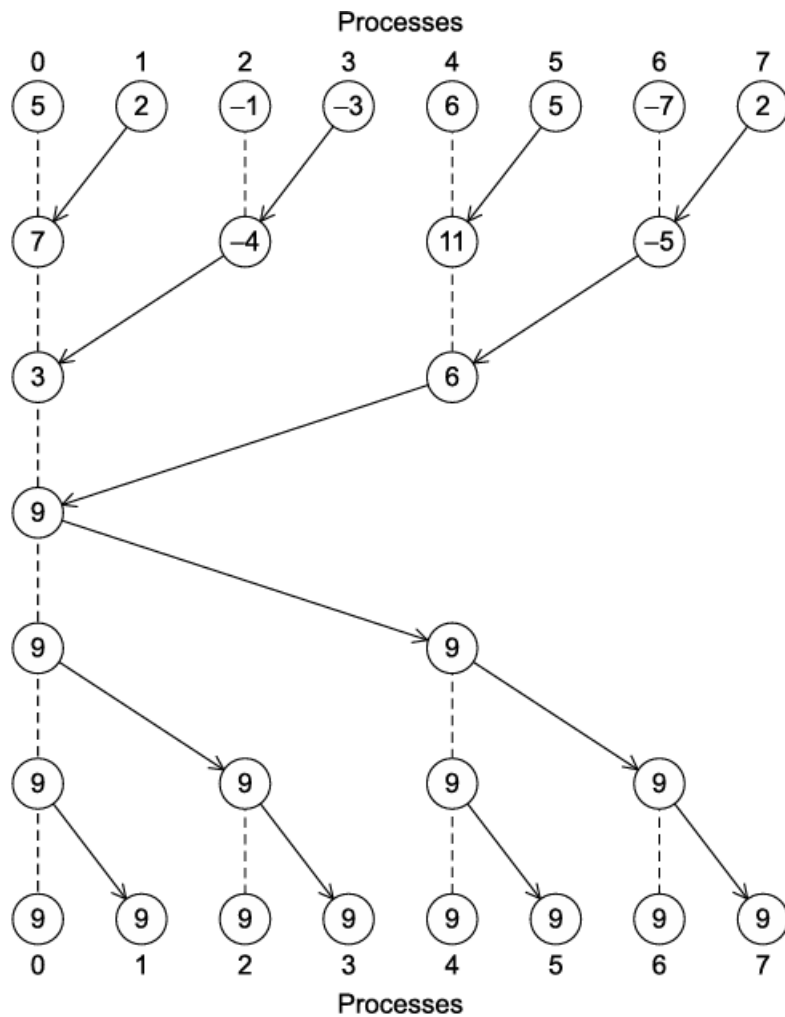
3.4.4 MPI_Allreduce

In some parallel programs, **all processes may need access to the result of a global computation**, such as a sum. While MPI_Reduce provides the result only to a designated destination process, **MPI_Allreduce ensures that the result is distributed to all processes in the communicator**. This eliminates the need to implement custom distribution schemes, such as reversing the branches of a tree structure (as in Fig. 3.8) or designing a **butterfly communication pattern** (Fig. 3.9), which could be complex to code and optimize.

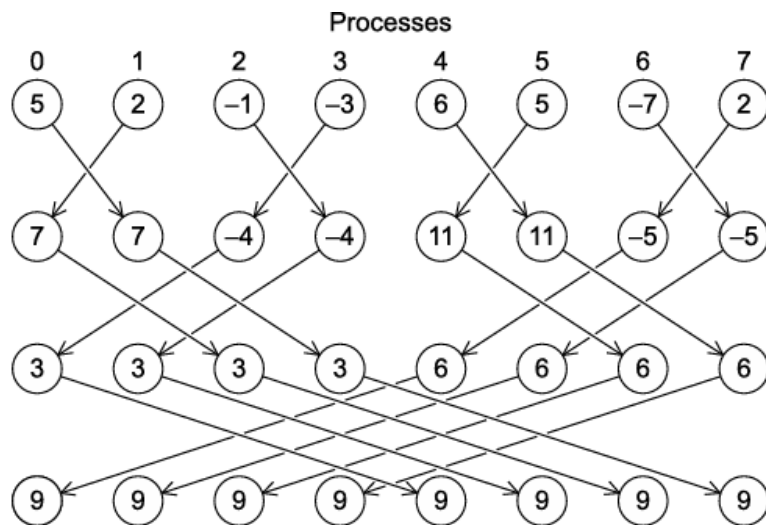
MPI handles this efficiently through the following function:

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

The parameters are identical to those of MPI_Reduce, except that **there is no dest_process**—because **every process receives the result**. This function simplifies parallel programming by removing the burden of manually distributing aggregated data across processes and allows all processes to **proceed with further computation using the shared result**.

**FIGURE 3.8**

A global sum followed by distribution of the result.

**FIGURE 3.9**

A butterfly-structured global sum.

3.4.5 Broadcast

To optimize **input distribution** in a parallel program—similar to how we optimized the global sum using tree-structured communication—we can use **broadcasting**. A **broadcast** is a collective communication operation where **data from a single process (the source)** is **sent to all other processes** in the communicator. For example, if process 0 reads the input data, it can **broadcast** the values of *a*, *b*, and *n* to all other processes using a tree structure like the one in **Fig. 3.10**.

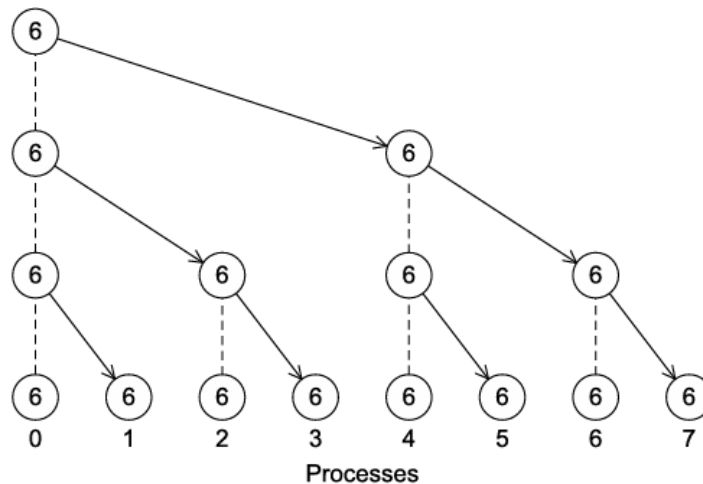


FIGURE 3.10

A tree-structured broadcast.

MPI provides the `MPI_Bcast` function for this purpose:

```

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

```

In this function, the process identified by `source_proc` sends data to all processes in the communicator `comm`. The `data_p` parameter is an **input** on the source process and an **output** on all others. Although labeled **in/out**, its role depends on the process's rank.

Using `MPI_Bcast`, we **avoid writing separate send operations to each process** and benefit from **optimized implementations** provided by MPI—just like with `MPI_Reduce`.

```

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.

3.4.7 Scatter

When testing a **vector addition function** in MPI, it's helpful to input the dimension and elements of vectors *x* and *y*. While **MPI_Bcast** works well for broadcasting the vector size, using it to broadcast entire vectors would be inefficient. For example, with 10 processes and vectors of 10,000 elements, broadcasting the full vectors would force every process to allocate memory for 10,000 elements—despite only working on a subset.

To solve this, MPI offers **MPI_Scatter**, which allows **process 0 to distribute only the required segment** of a vector to each process. This enables each process to **store and operate only on its assigned subvector**, greatly optimizing memory usage. The function signature is:

```

int MPI_Scatter(
    void* send_buf_p,    // in
    int send_count,      // in (per process)
    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 using a **block distribution** and if *n* is divisible by *comm_sz*, then *send_count* and *recv_count* should both be *local_n = n / comm_sz*, and *send_type / recv_type* should be *MPI_DOUBLE*. Each process receives its *local_n* chunk from *send_buf_p*, while process 0 reads and holds the entire input vector. The *MPI_Scatter* call sends the **first block to process 0, second to process 1, and so on**, making it ideal for **block distributions** only. This approach is used in the *Read_vector* function 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.

A key caveat: this strategy **only works efficiently when n is evenly divisible by comm_sz** and block distribution is used.

3.4.8 Gather

To make a vector addition program useful, we need a way to **view the results**. Since the resulting vector is distributed across processes, we need to **collect all subvector results onto process 0**, where they can be printed in full. MPI provides the collective function **MPI_Gather** for this purpose. The function allows each process to send its portion of the vector to a single destination process (typically rank 0), which stores the data in the correct order.

The function prototype is:

```

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
);

```

Each process sends the contents of `send_buf_p` to `dest_proc`. On the destination process (e.g., process 0), these chunks are stored in `recv_buf_p` in **rank order**: the data from **process 0 goes into the first block**, from **process 1 into the second**, and so on. So, if using **block distribution** with each process holding `local_n` elements, then `recv_count` should also be `local_n`. Importantly, `recv_count` refers to the number of elements **from each process, not the total number** collected.

This is effectively the **reverse operation of MPI_Scatter**, and similar constraints apply: it **assumes all blocks are of equal size** and works best with **block-distributed vectors**. The gathering logic for printing a distributed vector is implemented in **Program 3.10**.

```

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.

3.4.9 Allgather

To perform **parallel matrix-vector multiplication**, we begin with the mathematical formulation: if A is an $m \times n$ matrix and x is a vector of n components, then $y = Ax$ results in a vector y of m components, where $y_i = a_{i0}x_0 + a_{i1}x_1 + \dots + a_{i,n-1}x_{n-1}$. To parallelize this, we use **block row distribution** for matrix A so that each process gets a set of rows, and the corresponding **block distribution of y** so that process q computes the values of y for its rows of A .

The challenge arises with vector x , since each component of y requires **all components of x** . In a serial code, this is straightforward, but in parallel code—especially in **iterative applications** where y becomes the next x —we typically use the **same block distribution** for both x and y . To compute the result, each process needs the **entire vector x** , not just its block.

One naïve approach is to use **MPI_Gather** followed by **MPI_Bcast**, but this introduces redundant communication. Instead, **MPI provides MPI_Allgather**, a collective operation that gathers data from all processes and distributes the complete result to all processes in a single call.

```
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 collects `send_buf_p` from each process and **concatenates the segments** into `recv_buf_p` on **every process**. Each process ends up with a **full copy of the distributed vector**, enabling them to compute their dot-products locally. Typically, `send_count = recv_count`, and the function is efficient for **repeated calls**, such as in iterative solvers, where `x` is reused. For performance, it's advisable to allocate `x` once in the **calling function** and pass it to the matrix-vector multiplication routine, as done in **Program 3.12**.

```
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.

3.5 MPI-derived Datatypes

In **distributed-memory systems**, communication is significantly more **costly** than local computation. For instance, sending a double between nodes is far slower than a local addition. Moreover, sending many small messages is much less efficient than sending one large message. As demonstrated in the example with double `x[1000]`, a loop sending individual elements takes **50–100× longer** than sending the entire array in a single call. To **reduce message overhead**, MPI supports three strategies: using the **count** parameter in communication functions, **derived datatypes**, and **MPI_Pack/Unpack**.

Derived datatypes allow grouping multiple memory elements—including those of different types—into a single transferable unit. For instance, in the trapezoidal rule program, three separate `MPI_Bcast` calls (for `a`, `b`, and `n`) can be replaced with a single call using a derived

datatype composed of **two MPI_DOUBLES** and **one MPI_INT**. This requires knowledge of the **displacements** of variables in memory. Suppose the addresses of a, b, and n are 24, 40, and 48 respectively. Then the derived type is defined by:

```
{(MPI_DOUBLE, 0), (MPI_DOUBLE, 16), (MPI_INT, 24)}
```

To construct this derived datatype, MPI provides:

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

For our example:

```
c
int array_of_blocklengths[3] = {1, 1, 1};
MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};
```

Displacements can be computed using:

```
c
MPI_Get_address(&a, &a_addr);
MPI_Get_address(&b, &b_addr);
MPI_Get_address(&n, &n_addr);
```

Then:

```
c
array_of_displacements[0] = 0;
array_of_displacements[1] = b_addr - a_addr;
array_of_displacements[2] = n_addr - a_addr;
```

After building the datatype using `MPI_Type_create_struct`, we must **commit** it with:

```
c
MPI_Type_commit(&input_mpi_t);
```

And then we can broadcast all three values at once:

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

When done, we **free** the derived type using:

```
c
MPI_Type_free(&input_mpi_t);
```

This method simplifies communication and enhances performance. These steps are implemented in a reusable `Build_mpi_type` function, which is called from an updated `Get_input` routine (see **Program 3.13**). This technique is highly useful for compact communication in **heterogeneous data structures**.

```

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.

3.6 Performance Evaluation of MPI Programs

One of the main reasons we write **parallel MPI programs** is to achieve better performance compared to serial versions. But how do we measure this improvement? The key is to evaluate **only the computational part** (like matrix-vector multiplication), ignoring unrelated tasks like input and output.

MPI provides a built-in timing function called **MPI_Wtime()**, which returns the **elapsed wall clock time** (real-world time, including wait and idle periods). This differs from CPU time (returned by functions like clock()), which doesn't account for waiting during communication.

3.6.1 Taking Timings

When timing MPI code, we typically **exclude non-computational steps** (e.g., input, output). We use **MPI_Wtime()** as follows:

```
double start, finish;
start = MPI_Wtime();
/* Code to be timed */
finish = MPI_Wtime();
printf("Proc %d > Elapsed time = %e seconds\n", my_rank, finish - start);
```

For **serial code**, MPI is not needed. We can use the **GET_TIME macro** (defined in timer.h, downloadable from the book's website) which wraps gettimeofday():

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

Be aware that GET_TIME is a **macro**, not a function, and takes a **double** (not a pointer). Also, if timer.h is not in your working directory, compile with:

```
gcc -g -Wall -I/home/user/my_include -o prog prog.c
```

Both **MPI_Wtime()** and **GET_TIME** report **wall clock time**, which includes communication delays (e.g., time spent waiting in MPI_Recv). This is more relevant in parallel environments where idle time impacts real performance.

To obtain a **single elapsed time** for the parallel run (instead of separate times for each process), we use **MPI_Barrier** to synchronize processes and **MPI_Reduce** with the **MPI_MAX** operator to gather the **maximum elapsed time**:

```
double local_start, local_finish, local_elapsed, elapsed;
MPI_Barrier(comm); // Synchronize all processes
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);
```

This gives a more realistic **total parallel execution time**.

Also, we must account for **variability** in timings due to **OS activity** and other system-level factors. Even with identical input and system load, execution time may vary across runs. Hence, we usually report the **minimum observed runtime**, assuming the fastest run occurred with the least interference.

When running MPI on **hybrid multicore systems**, it's often more efficient to run **one MPI process per node**, reducing communication overhead and contention, leading to more consistent and improved timings.

3.6.2 Results

The **timing results** for the **matrix-vector multiplication** program are summarized in **Table 3.5**. The input matrices are **square**, and times are shown in **milliseconds**, rounded to **two significant digits**.

- For **comm_sz = 1**, the timings represent a **serial run** on a single core.
- As expected, for a fixed number of processes, increasing the matrix size **n** leads to longer runtimes.
- For **small numbers of processes**, doubling **n** roughly **quadruples** the runtime. But for **larger process counts**, this pattern does not always hold.

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

comm_sz	Order of Matrix				
	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** (number of processes):

- **Runtime usually decreases**, especially for **large values of n**.
- Doubling the number of processes roughly **halves the runtime**.
- But for **small n**, adding more processes offers **diminishing or no benefit**.

For instance:

```
T_parallel(1024, 8) ≈ T_parallel(1024, 16)
```

This behavior is **typical** of parallel programs. As **problem size increases**, runtimes increase **regardless of the number of processes**, though the **rate of increase** depends on the number of processes:

- **One-process times** increase steadily.
- **Multi-process times** can fluctuate (especially at 16 processes).

A useful performance model expresses **parallel time** as:

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

Where:

- **$T_{\text{serial}}(n)$** is the time taken by the **serial program** on input size **n** .
- **$T_{\text{parallel}}(n, p)$** is the time taken by the **parallel program** on input size **n** using **p** processes.
- **T_{overhead}** represents the **parallelization cost**, often due to **communication**, especially in **MPI programs**.

For the matrix-vector multiplication, the serial code has two nested 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];
}
```

This performs:

- **$2n$** floating point operations per row (n multiplications + n additions),
- Executed **m times**, giving **$2mn$** total floating point operations. So, if **$m = n$** , the serial runtime can be approximated as:

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

(for some constant **a**). In

the **parallel version**:

- Each process handles **$(n/p) \times n$** elements.
- So the local computation time is approximately:

$$T_{\text{local_compute}} = a * n^2 / p$$

But there's an additional overhead due to **MPI_Allgather**, so the parallel time becomes:

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

Empirical Observations

For **small p (2 or 4)** and **large n** , the runtime behaves close to **$T_{\text{serial}}(n) / p$** : Doubling **p** approximately halves the runtime:

```
T_serial(4096) ≈ 1.9 × T_parallel(4096, 2)
T_parallel(8192, 2) ≈ 2.0 × T_parallel(8192, 4)
```

Doubling **n** roughly quadruples the time:

```
SCSS

T_serial(4096) ≈ 4.0 × T_serial(2048)
T_parallel(4096, 2) ≈ 3.9 × T_parallel(2048, 2)
```

Thus, the **MPI_Allgather overhead** appears to have **little impact** when:

- **n is large**, and **p is small**.

Breakdown for Small **n** and Large **p**

However, the approximation **fails** when:

- **n is small**, and **p is large**.

For example:

```
T_parallel(1024, 8) = T_parallel(1024, 16)
T_parallel(2048, 16) = 1.5 × T_parallel(1024, 16)
```

This indicates that for **small problem sizes**, the **communication cost** (**T_allgather**) dominates the performance, limiting scalability.

3.6.3 Speedup and Efficiency

An essential metric in evaluating parallel program performance is **speedup**, denoted as:

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

This represents how many times **faster** the parallel program runs with **p processes** compared to the serial version on input of size **n**. The **ideal case** is when:

$$S(n, p) = p$$

This is called **linear speedup**, indicating perfect scaling—i.e., a program with 16 processes is 16 times faster than the serial one. However, **linear speedup is rare in practice** due to communication overhead and load imbalance.

In the case of the **matrix-vector multiplication program**, the **speedups** obtained are listed in **Table 3.6**. Observations include:

Table 3.6 Speedups of parallel matrix-vector multiplication.

comm_sz	Order of Matrix				
	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

- For **small p** and **large n**, the program achieved **near-linear speedup**.
- For **large p** and **small n**, the speedup was **significantly less** than p.
- The **worst performance** occurred at **n = 1024** with **p = 16**, where the speedup dropped to **only 2.4**, far from the ideal value of 16.

Another related metric is **parallel efficiency**, defined as the **speedup per process**:

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

A value of **1.0** indicates **perfect efficiency**, where each process contributes fully to the speedup. In general, efficiency is **less than 1**, and decreases as **communication and synchronization overheads** increase.

The **efficiencies** for matrix-vector multiplication are shown in **Table 3.7**:

- When **n is large** and **p is small**, efficiency is **close to 1.0**.
- As **p increases** and **n remains small**, efficiency drops **significantly**.

This reinforces the principle that **parallelism pays off** when applied to **large problems** and that adding more processes to small problems often results in **poor scalability**.

Table 3.7 Efficiencies of parallel matrix-vector multiplication.

comm_sz	Order of Matrix				
	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

3.6.4 Scalability

When analyzing the performance of parallel programs, it's not enough to look only at **speedup** and **efficiency**. Another crucial factor is **scalability**.

A parallel program is said to be **scalable** if the **problem size can be increased at a rate that maintains the same efficiency** as the number of processes (**p**) increases. In other words, a **scalable program keeps working well** even when we **use more processors**, provided we **increase the workload proportionally**.

However, this definition has some ambiguity—specifically in the phrase: “*problem size can be increased at a rate...*” Let's examine two hypothetical programs:

- **Program A** maintains a **constant efficiency of 0.75** for all $p \geq 2$, **regardless of problem size**.
- **Program B** has efficiency defined as:

$$E(n, p) = \frac{n}{625p}, \quad \text{for } p \geq 2 \text{ and } 1000 \leq n \leq 625p$$

In Program B, if we **increase both n and p** proportionally, we can **maintain constant efficiency**. For example:

- When $n = 1000, p = 2$, then $E = \frac{1000}{1250} = 0.8$
- If we double both: $n = 2000, p = 4$, then $E = \frac{2000}{2500} = 0.8$

So **both programs are technically scalable**, but **Program A is more scalable** because it does not even require an increase in problem size to maintain efficiency. This leads to two common categories of scalability:

- **Strong scalability**: Constant efficiency **without** increasing problem size (Program A).
- **Weak scalability**: Constant efficiency **if** the problem size increases proportionally with the number of processes (Program B).

Matrix-Vector Multiplication and Scalability

Looking at the **parallel efficiency values in Table 3.7**, we can conclude:

- The matrix-vector multiplication program is **not strongly scalable**. For nearly every case, **increasing the number of processes results in reduced efficiency**.
- However, the program shows traits of **weak scalability**. When both **p** and **n** are **doubled, parallel efficiency often improves**.

For instance:

- Increasing **p** from 4 to 8 or 8 to 16 **with a corresponding increase in n** consistently improves or maintains efficiency.
- The **only slight deviation** happens when $p = 2$ and $p = 4$, but this is usually not a concern, since scalability discussions often focus on **larger processor counts**.

Thus, the **matrix-vector multiplication program is weakly scalable**. While not ideal for small problems or low processor counts, it **handles larger data and processor configurations reasonably well**—a **desirable property** in large-scale parallel computing.

3.7 A Parallel Sorting Algorithm

In a **distributed memory environment**, a *parallel sorting algorithm* deals with data (keys) spread across multiple processes. The **input and output** depend on the data distribution— keys can either be **distributed among processes** or **assigned to a single process**. In this case, we focus on an algorithm where both **input and output are distributed**, meaning each process starts and ends with $\frac{n}{p}$ keys, assuming n is divisible by the number of processes $p = \text{comm_sz}$. Initially, keys are **randomly distributed** across processes. The goal is to ensure that, after execution:

- Each process's local keys are **sorted in increasing order**, and
- For any two processes $q < r$, **every key in process q is less than or equal to every key in process r** .

Thus, if we concatenate the keys from all processes in order of increasing rank, we get a **fully sorted list**. For simplicity, we assume the keys are of type `int`. This sets the stage for designing efficient and scalable **parallel sorting algorithms** that operate correctly and coherently in a **distributed setting**.

3.7.1 Some Simple Serial Sorting Algorithms

To better understand parallel sorting, let's briefly review a couple of **serial sorting techniques**. One well-known method is **Bubble Sort**, which repeatedly compares and swaps adjacent values to move larger elements toward the end of the list. On each outer loop pass, the largest unsorted value "bubbles" into its correct position, and the unsorted portion shrinks by one.

However, **Bubble Sort is inherently sequential**—a swap at one position may affect adjacent comparisons. For instance, a sequence like 9, 5, 7 needs ordered swaps to correctly become 5, 7, 9; swapping 5 and 7 prematurely would yield the wrong result.

```
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.

To introduce **parallelism**, we turn to **Odd-Even Transposition Sort**, which breaks dependencies using phases:

- **Even phases** compare and swap: (a[0], a[1]), (a[2], a[3]), ...
- **Odd phases** compare and swap: (a[1], a[2]), (a[3], a[4]), ...

Example:

```
Start:      5, 9, 4, 3
Even phase: 5, 9, 3, 4
Odd phase:  5, 3, 9, 4
Even phase: 3, 5, 4, 9
Odd phase:  3, 4, 5, 9 (Sorted)
```

A theorem guarantees that n elements will be sorted in at most n phases.

3.7.2 Parallel Odd-Even Transposition Sort

In a **distributed memory system**, the same concept is extended across p processes, each holding n/p elements (assuming n is evenly divisible by p).

When $n=p$ (each process has one key), the algorithm is intuitive: At each phase, process i exchanges keys with either $i-1$ or $i+1$ based on phase type (even or odd), and keeps either the **smaller** or **larger** key based on its rank.

But this is impractical for real applications (we usually have $n > p$), and communication overhead outweighs benefits for small inputs.

```
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.

When each process holds multiple keys, the process is:

1. **Locally sort keys** using fast serial sort like qsort.

2. In each phase:

- Determine a **partner** (left or right neighbor depending on phase).
- **Exchange full key arrays** with the partner.
- **Merge the two arrays:**
 - Lower-ranked process keeps the **smaller half**.
 - Higher-ranked process keeps the **larger half**.

This ensures that:

- Each process maintains sorted keys.
- Global ordering progresses toward a fully sorted distributed list.

Table 3.8 illustrates this procedure across 4 processes and 16 keys.

Table 3.8 Parallel odd-even transposition sort.

Time	Process			
	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

Theorem

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

Pseudocode

```
Sort local keys;
for (phase = 0; phase < comm_sz; phase++) {
    partner = Compute_partner(phase, my_rank);
    if (partner != MPI_PROC_NULL) {
        Send my keys to partner;
        Receive partner's keys;
        if (my_rank < partner)
            Keep smaller keys;
        else
            Keep larger keys;
    }
}
```

Computing the Partner:

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

if (partner < 0 || partner >= comm_sz)
    partner = MPI_PROC_NULL;
```

MPI_PROC_NULL is a **special constant** indicating no communication should occur for that process in that phase.

3.7.3 Safety in MPI Programs

In a parallel MPI program, communication between processes is typically implemented using MPI_Send and MPI_Recv calls. For example:

```
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);
```

However, this approach can lead to problems such as **program hanging or crashing**, especially when processes engage in mutual communication simultaneously. The behavior of MPI_Send is defined by the MPI standard to be either buffered (non-blocking) or blocking. Buffered mode allows the program to proceed without waiting for a corresponding receive, while blocking mode waits until a matching MPI_Recv is initiated.

Most MPI implementations use a **threshold** mechanism: small messages are typically buffered, whereas larger messages invoke blocking behavior. When all processes perform MPI_Send simultaneously with large messages, none may reach the MPI_Recv stage, leading to a **deadlock**—a situation where every process is waiting indefinitely for another to proceed.

A program that implicitly assumes buffered communication is termed an **unsafe MPI program**. While such a program might execute correctly for small problem sizes, it is prone to failure when the message sizes increase, or system buffering behavior changes.

Detecting Unsafe Communication

To determine whether a program is safe, the MPI standard provides a synchronous version of MPI_Send called MPI_Ssend. The call to MPI_Ssend ensures that the send operation blocks until the matching receive is initiated by the receiving process.

```
MPI_Ssend(my_keys, n / comm_sz, MPI_INT, partner, 0, comm);
```

By substituting MPI_Send with MPI_Ssend, one can check the safety of the program. If the program executes correctly with MPI_Ssend, it is considered safe. Otherwise, if it hangs or crashes, the original version using MPI_Send is unsafe.

Causes of Unsafe Programs and Restructuring

Unsafe communication often arises when all processes attempt to **send first and receive later**. For instance, in a ring communication pattern, where process q sends to $(q + 1) \% \text{comm_sz}$, simultaneous send operations can result in a deadlock:

```
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);
```

To make such communication patterns safe, the communication must be **restructured**, so that some processes receive before sending. An example of a safe structure is:

```
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);
}
```

This pattern ensures that **half of the processes perform Recv before Send**, breaking the circular dependency that leads to deadlock. This method is effective for both **even and odd values of comm_sz**, as shown in **Figure 3.13**, which illustrates how a safe communication order can be maintained even in an odd-sized process ring.

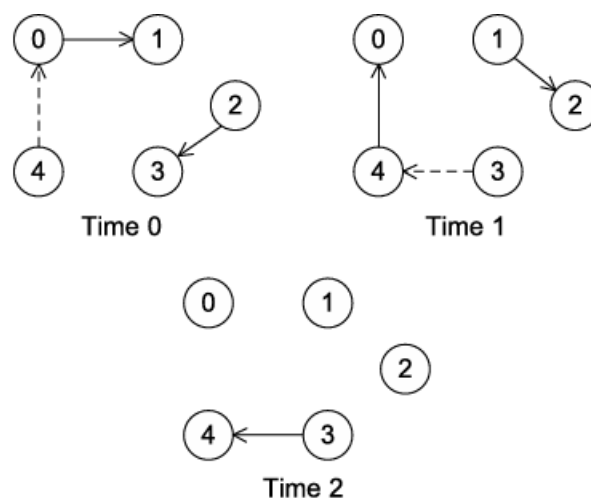


FIGURE 3.13

Safe communication with five processes.

Using MPI_Sendrecv for Safe Communication

An alternative and cleaner approach to avoid explicit coordination is to use the `MPI_Sendrecv` function, which performs a **send and receive operation in a single call**, ensuring safe execution:

```
MPI_Sendrecv(send_buf, send_size, send_type, dest, send_tag,
             recv_buf, recv_size, recv_type, source, recv_tag,
             comm, &status);
```

lly schedules the

communication to ensure that one side sends while the other receives.

Additionally, if the **send and receive buffers are the same**, MPI provides:

```
MPI_Sendrecv_replace(buf, buf_size, buf_type, dest, send_tag,
                    source, recv_tag, comm, &status);
```

This function sends the data in `buf`, replaces it with the received data, and is useful for **in-place communication patterns**, such as in odd-even transposition sort.

3.7.4 Final Details of Parallel Odd-Even Sort

The final implementation of the parallel odd-even transposition sort uses a safe communication strategy by combining the send and receive operations into a single call to `MPI_Sendrecv`. This helps avoid **deadlock** issues that arise from unsafe usage of `MPI_Send` and `MPI_Recv`, particularly when all processes attempt to send first. The improved communication structure ensures reliable exchange of keys between paired processes during each phase. The general algorithm remains the same: each process sorts its local keys first, and during each phase, communicates with a partner process, compares the combined keys, and retains either the smaller or larger half depending on its rank.

To make the process efficient, we avoid sorting the full list of combined keys. Instead, we **merge** the two sorted sublists—each of size n/p —to select the required subset. If a process is supposed to keep the smaller keys, it performs a forward merge and stops once n/p keys are obtained. If it must keep the larger keys, it merges backward from the end of the arrays. This **partial merge** is faster than full sorting. An additional **optimization** skips the need to copy arrays entirely by simply **swapping pointers**, which reduces memory operations (this pointer-swapping trick is elaborated in Exercise 3.28).

The **performance** of this final version is reflected in **Table 3.9**, which shows the run-times. It's important to note that for a single process, the sorting is done using **serial quicksort**, which is significantly faster than a serial odd-even sort. As expected, run-times improve with increasing number of processes, especially when n is large enough to offset communication costs. However, for smaller n , gains are modest.

This final version represents a practical and scalable parallel sorting strategy for **distributed memory systems**, leveraging **local sorting, partner communication, safe message passing, and efficient merging**.

```

void Merge_low(
    int  my_keys[],      /* in/out  */
    int  recv_keys[],   /* in      */
    int  temp_keys[],   /* scratch */
    int  local_n        /* = n/p, in */) {
    int m_i, r_i, t_i;

    m_i = r_i = t_i = 0;
    while (t_i < local_n) {
        if (my_keys[m_i] <= recv_keys[r_i]) {
            temp_keys[t_i] = my_keys[m_i];
            t_i++; m_i++;
        } else {
            temp_keys[t_i] = recv_keys[r_i];
            t_i++; r_i++;
        }
    }

    for (m_i = 0; m_i < local_n; m_i++)
        my_keys[m_i] = temp_keys[m_i];
} /* Merge_low */

```

Program 3.16: The Merge_low function in parallel odd-even transposition sort.

Table 3.9 Run-times of parallel odd-even sort (times are in milliseconds).

Processes	Number of Keys (in thousands)				
	200	400	800	1600	3200
1	88	190	390	830	1800
2	43	91	190	410	860
4	22	46	96	200	430
8	12	24	51	110	220
16	7.5	14	29	60	130

MODULE-4

Shared-memory programming with OpenMP – OpenMP pragmas and directives, the trapezoidal rule, scope of variables, the reduction clause, loop carried dependency, scheduling, producers and consumers, caches, cache coherence and false sharing in OpenMP, tasking, and thread safety.

Introduction to OpenMP

OpenMP is an **API (Application Programming Interface)** designed specifically for **shared-memory MIMD (Multiple Instruction, Multiple Data)** programming. The term “MP” stands for **multiprocessing**, aligning with the shared-memory computing paradigm. In systems targeted by OpenMP, **multiple autonomous CPU cores** can access a **common main memory**, as typically illustrated in a shared-memory architecture diagram (like Fig. 5.1).

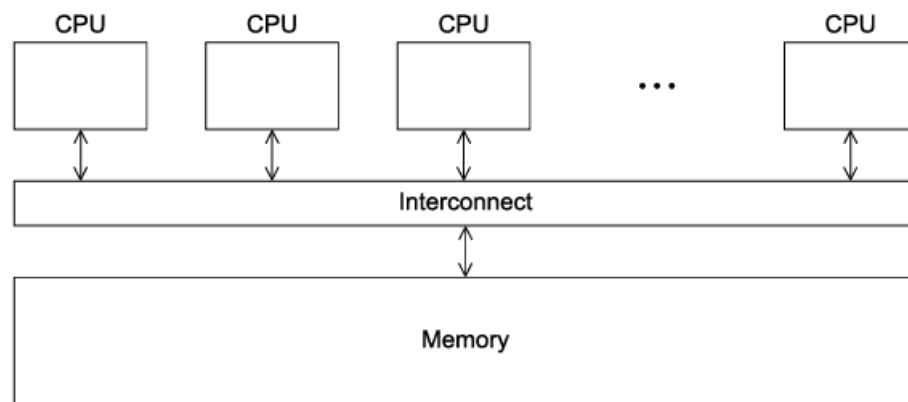


FIGURE 5.1

A shared-memory system.

Comparison with Pthreads

Although both **OpenMP** and **Pthreads** serve the purpose of shared-memory programming, they differ in their approach and complexity:

- **Pthreads is low-level** and requires **explicit thread management**. The programmer defines the behavior of each thread manually.
- **OpenMP offers a higher-level abstraction**. The programmer can simply indicate that a block of code should run in parallel. The **OpenMP compiler and runtime system automatically manage thread behavior**.

This difference makes **OpenMP easier to use** for certain types of problems, especially those involving structured parallelism.

- **Pthreads is a library-based API**, and any standard C compiler can compile Pthreads code, provided the system supports the library.

- **OpenMP requires compiler support.** Not all compilers support OpenMP, which means you may encounter compilers that cannot handle OpenMP directives.

The dual existence of OpenMP and Pthreads highlights a **trade-off**:

- **Pthreads offers full control** and is suitable for complex, custom thread behaviors.
- **OpenMP is simpler to use** but may **limit low-level control** over thread interactions.

This makes OpenMP suitable for rapid development and easy parallelization of certain serial codes, while Pthreads is preferred for **fine-grained, customized concurrency control**.

OpenMP was developed by a consortium of programmers and researchers who aimed to **simplify high-performance parallel programming**. Writing large-scale programs using lower-level APIs like Pthreads or MPI can be challenging and error-prone. OpenMP was designed to **allow incremental parallelization** of existing serial programs, a feature not possible with MPI and only partly feasible with Pthreads.

4.1 Getting started

OpenMP uses a **directives-based shared-memory API**, primarily through **compiler directives called pragmas** in C and C++. **Pragmas** are special preprocessor instructions that **enable behavior beyond the standard C language**, and **compilers that don't support them simply ignore them**. This makes OpenMP programs **backward compatible**—they can still be compiled and run sequentially on systems without OpenMP support, although without parallelism.

In C/C++, **OpenMP pragmas begin with**

#pragma,

where the # must appear at the **start of the line**, and the rest of the directive should be **indented consistently** with the code. By default, **pragmas are single-line instructions**, so if a directive needs to span multiple lines, **the newline must be escaped using a backslash** \.

This design makes OpenMP flexible and safe for integration into existing C programs. For instance, a simple OpenMP “hello, world” program (see **Program 5.1**) can include a #pragma omp parallel directive, and **if OpenMP is not supported**, the compiler will ignore the directive and compile the code as a regular sequential program.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5  void Hello(void); /* Thread function */
6
7  int main(int argc, char* argv[]) {
8      /* Get number of threads from command line */
9      int thread_count = strtol(argv[1], NULL, 10);
10
11     #pragma omp parallel num_threads(thread_count)
12     Hello();
13
14     return 0;
15 } /* main */
16
17 void Hello(void) {
18     int my_rank = omp_get_thread_num();
19     int thread_count = omp_get_num_threads();
20
21     printf("Hello from thread %d of %d\n",
22           my_rank, thread_count);
23
24 } /* Hello */

```

Program 5.1: A “hello, world” program that uses OpenMP.

4.1.1 Compiling and running OpenMP programs

To compile an OpenMP program using **GCC**, we must include the **-fopenmp** option. For example, a typical compilation command would be:

```
$ gcc -g -Wall -fopenmp -o omp_hello omp_hello.c
```

This tells the compiler to enable OpenMP support during compilation. When running the program, we can specify the **number of threads** we want to use by passing it as a command-line argument. For instance, to run with **four threads**, we use:

```
$ ./omp_hello 4
```

The output might look like:

```

cpp

Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4

```

However, since all threads are writing to **stdout concurrently**, the **output order is not guaranteed**. The messages may appear in any **permutation of thread ranks**, such as:

```
Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
```

To test the program with **just one thread**, we can run:

```
bash
$ ./omp_hello 1
```

And the expected output will be:

```
Hello from thread 0 of 1
```

4.1.2 The program

In OpenMP programs, in addition to using **directives**, we also include a library of functions and macros by adding the header file:

```
#include <omp.h>
```

This is essential for calling OpenMP-specific functions like `omp_get_thread_num()` and `omp_get_num_threads()`.

Like Pthreads, OpenMP programs often **take the number of threads as a command-line argument**. This is typically done using the `strtol()` function, which converts the argument from string to integer format. The syntax of this function is:

```
long strtol(const char* str, char** endptr, int base);
```

In most cases, we pass NULL as the second parameter and use 10 as the base for decimal numbers.

When a C program begins execution, it runs in a **single-threaded process**. However, when the OpenMP **parallel directive** is encountered, the compiler spawns **multiple threads**, each of which can execute a **structured block** (a single entry and exit point). The simplest directive is:

```
#pragma omp parallel
```



FIGURE 5.2

A process forking and joining two threads.

This causes the runtime system to start a **team of threads**, including the **master thread** (thread 0) and other threads it spawns (child threads). The number of threads used is determined by the system, unless the programmer overrides it using a clause:

```
#pragma omp parallel num_threads(thread_count)
```

The actual number of threads started may vary depending on system limits, but most systems support **hundreds or thousands of threads**.

Each thread in the team runs the specified block of code. For instance, if the block calls a function like Hello(), each thread executes it. When all threads complete this block, an **implicit barrier** ensures that they wait for each other before the parent thread proceeds. The **parent thread** is the one that encountered the parallel directive and started the team. In most cases, it is also the master thread.

Each thread has its own **private stack**, meaning it will create its **own local variables** when executing the function. To retrieve its **thread ID** and the **total number of threads**, a thread can call:

```
int omp_get_thread_num(void);
int omp_get_num_threads(void);
```

These functions return the thread's rank (from 0 to thread_count-1) and the total number of active threads, respectively.

All threads share access to **stdout**, so each thread can print its message using printf. However, because output is not synchronized, the **order of printed lines is nondeterministic**. This behavior demonstrates **OpenMP's implicit parallelism**, where we achieve substantial multithreaded execution with minimal code, contrasting with the more verbose and explicit thread management required in Pthreads

4.1.3 Error checking

In the provided OpenMP example program, **no error checking is performed** to keep the code simple and compact. However, in real-world scenarios, this is **risky** and **not recommended**. Good programming practice dictates that developers must anticipate and handle **possible errors**, especially those involving **user input and system support**.

A primary check involves ensuring that a **valid command-line argument** is supplied. After retrieving the number of threads using strtol(), we should verify that the value is **positive**, as negative or zero threads would be invalid. Furthermore, although not critical in simple programs, one could also confirm whether the **actual number of threads** created by OpenMP matches the user-supplied count.

Another source of potential errors is the **compiler's support for OpenMP**. If OpenMP is **not supported**, the compiler may silently **ignore the #pragma omp parallel directive**, but still generate **errors** for #include <omp.h> or calls to OpenMP functions like omp_get_thread_num() and omp_get_num_threads().

To handle this, we can **conditionally compile** OpenMP-specific code by checking whether the **macro `_OPENMP` is defined**. This macro is automatically defined by compilers that support OpenMP. For example, instead of directly including the header:

```
#include <omp.h>
```

We use:

```
c
#ifdef _OPENMP
# include <omp.h>
#endif
```

Similarly, function calls that depend on OpenMP can be wrapped as:

```
c
#ifdef _OPENMP
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();
#else
    int my_rank = 0;
    int thread_count = 1;
#endif
```

This fallback ensures that **even if OpenMP is not supported**, the program can still **execute sequentially** with a **single thread**, assuming rank 0 and a thread count of 1.

The authors note that a more robust version of the sample program (with these checks) is available on the book's website. However, in textbook examples, they typically **omit error checks** for clarity, and **assume** that OpenMP support is available.

4.2 The Trapezoidal Rule

The **trapezoidal rule** is a **numerical method** for approximating the **area under a curve** $y=f(x)$, between the interval $[a,b]$. It works by **dividing the interval into n subintervals** and estimating the area over each subinterval using a **trapezoid**.

If $h = \frac{b-a}{n}$ and $x_i = a + i \cdot h$, then the approximation of the area is given by:

$$\text{Area} \approx h \left[\frac{f(x_0)}{2} + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + \frac{f(x_n)}{2} \right]$$

This rule provides a simple way to **estimate definite integrals**, especially when the **analytical integration is complex or not possible**.

```

/* 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;

```

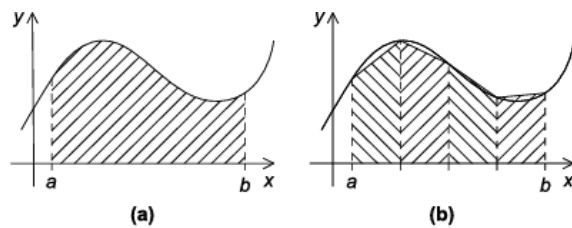


FIGURE 5.3

The trapezoidal rule.

4.2.1 A First OpenMP Version

1. **Identified two types of jobs:**
 - a. Computation of the areas of individual trapezoids
 - b. Adding the areas of trapezoids
2. **No communication** among jobs in the first collection (1a), but each job in (1a) communicates with job (1b).
3. **Assumption:** Number of trapezoids (n) is much larger than the number of cores. So, we **aggregated jobs** by assigning a **contiguous block of trapezoids** to each thread. This partitions the interval $[a, b]$ into **subintervals**, and each thread applies the **serial trapezoidal rule** to its subinterval (refer Figure 5.4).

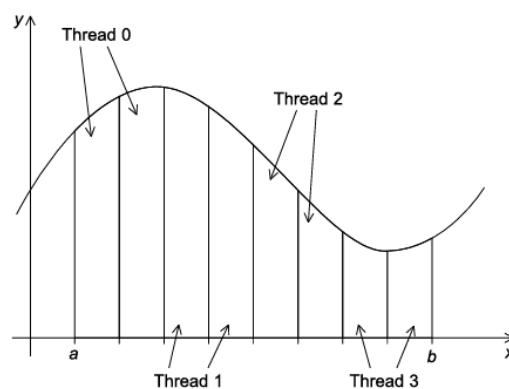


FIGURE 5.4

Assignment of trapezoids to threads.

A simple idea is to use a shared variable `global_result` and let each thread update it like this:

```
global_result += my_result;
```

But this leads to a **race condition** — if multiple threads execute this statement simultaneously, the result will be **unpredictable**.

Example scenario (with `my_result` = 1 for thread 0 and `my_result` = 2 for thread 1):

- Both threads load `global_result` = 0
- Each adds their `my_result`
- Final `global_result` ends up being 2, instead of expected 3 (See the **timetable table** in the text for detailed illustration)

This is a **race condition**:

- Multiple threads access a **shared resource**
- At least one of the accesses is an **update**
- Accesses **interleave** in a way that leads to incorrect results

The statement `global_result += my_result` is a **critical section** — code that updates a shared resource and must be executed by **only one thread at a time**.

In OpenMP, the **critical directive** ensures that only one thread at a time executes a particular block of code, providing **mutual exclusion** among threads. This prevents data races when multiple threads try to update shared variables simultaneously.

In **Program 5.2**, the **main function** first executes as a single thread to read the inputs — the number of threads, and the values of a , b , and n . The **parallel directive** at Line 17 specifies that the function **Trap** should be executed by *thread_count* threads in parallel. After the parallel region completes, all additional threads are terminated, and only the main thread continues to print the final result.

Within the **Trap function**, each thread obtains:

1. Its **rank** and the **total number of threads**,
2. The **base length** of the trapezoids,
3. The **number of trapezoids** assigned to it,
4. The **left and right endpoints** of its interval, and
5. Its **local contribution** to the shared variable `global_result`.

```

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.

Each thread computes its own **partial area** using local variables such as *local_a* and *local_b*, which differ for each thread. These local results are then added to **global_result** inside a **critical section**, ensuring that only one thread updates it at a time.

If n (the total number of trapezoids) is not **evenly divisible** by *thread_count*, the program will use fewer trapezoids than requested. For example, if $n = 14$ and *thread_count* = 4, each thread gets $local_n = 14 / 4 = 3$ trapezoids, resulting in only $4 \times 3 = 12$ trapezoids being used. Therefore, a check is included in the code to ensure that $n \% thread_count == 0$.

Since each thread handles a block of *local_n* trapezoids, the **interval width** for each thread is $local_n \times h$. Thus, the left and right endpoints for each thread are computed as:

$$local_a = a + my_rank \times local_n \times h$$

$$local_b = local_a + local_n \times h$$

This means, for example:

- Thread 0: $a + 0 \times local_n \times h$

- Thread 1: $a + 1 \times \text{local_n} \times h$
- Thread 2: $a + 2 \times \text{local_n} \times h$, and so on.

4.3 Scope Variable

the **scope of variables** decides which threads can access them during parallel execution.

In serial programming, a variable declared inside a function has *function scope*, while a variable declared outside all functions has *file scope*. Similarly, in OpenMP, variables can have **shared** or **private** scope depending on where they are declared.

A **shared variable** can be accessed by all threads in a team. Only one copy exists, and all threads operate on it. Variables declared **before** a parallel directive are shared by default. For example, in the trapezoidal rule program, variables like `a`, `b`, `n`, `global_result`, and `thread_count` are shared since they are declared in the main function before the parallel region.

A **private variable** is unique to each thread. Each thread has its own copy, and any modification by one thread does not affect others. Variables declared **inside** a parallel block or inside a function executed in parallel are private by default. For example, in the Trap function:

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

Here, `global_result_p` is a private pointer for each thread, but it refers to the **shared** variable `global_result` declared in main. Therefore, all threads update the same shared variable, and this requires synchronization using a critical section.

If `global_result` were private, each thread would have its own copy, and the final result in main would be incorrect.

4.4 The Reduction Clause

In parallel programs, when multiple threads update a shared variable (like `global_result`), we must ensure updates happen correctly without making the code sequential. Using a **critical section** ensures correctness but reduces parallelism since only one thread executes the section at a time.

To solve this, OpenMP provides the **reduction clause**, which allows each thread to have its own **private copy** of a variable and combine the results automatically after the parallel region ends.

Example without reduction:

```

cpp

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

```

Using reduction clause:

```

makefile

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

```

Here, `global_result` is declared as a reduction variable, and the operator `+` specifies that all threads' results are added together.

Syntax:

```

php-template

reduction(<operator> : <variable list>)

```

Valid operators in C: `+`, `*`, `-`, `&`, `|`, `^`, `&&`, `||`

OpenMP creates a **private copy** of the reduction variable for each thread.

Each thread updates its copy during execution.

At the end of the parallel block, OpenMP automatically **combines** all private results using the specified operator and stores the final value in the **shared variable**.

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

Operator	Identity Value
<code>+</code>	0
<code>*</code>	1
<code>-</code>	0
<code>&</code>	<code>~0</code>
<code> </code>	0
<code>^</code>	0
<code>&&</code>	1
<code> </code>	0

4.5 The *Parallel for* Directive

- The `#pragma omp parallel for` directive is used to automatically parallelize loops.
- It divides loop iterations among threads without needing manual work distribution.

```
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;
```

- The **parallel for** directive forks multiple threads to execute loop iterations in parallel.
- The **system divides iterations** (usually block-wise) among available threads.
- The **loop variable (i)** is **private** by default.
- Other variables are **shared** unless declared otherwise.
- The **reduction(+: approx)** clause ensures each thread keeps a private copy of `approx`, and all partial results are added safely at the end.
- Without reduction, `approx` would be a shared variable, causing incorrect results due to race conditions

4.5.1 Caveats

The `#pragma omp parallel for` directive makes it easy to parallelize loops — sometimes by simply adding a single line before a loop. However, there are a few important **limitations** to keep in mind:

- **Only for loops can be parallelized** — OpenMP does not support while or do-while loops directly.
(These can be rewritten as equivalent for loops if needed.)
- **Number of iterations must be known before loop execution.**
OpenMP can only parallelize loops when:
 - The iteration count can be determined **from the for statement itself**, and
 - **Before** the loop starts. or loops with break statements inside cannot be parallelized.

```
for (;;)
{
    // ...
}
```

- **Loop must be in canonical form.**
The loop should follow a standard pattern:

```
for (index = start; index < end; index += incr)
```

Restrictions:

- index must be an integer or pointer.
 - start, end, and incr must be compatible and constant during execution.
 - index can only be modified by the increment part of the for statement.
- **Structured block requirement.**
The loop should not have multiple exit points (like extra break or goto).
 - **Exception:**
The only allowed early exit is through a call to exit() inside the loop.

4.5.2 Data Dependences

When a for loop does not meet OpenMP's rules, the compiler may reject it. For example, consider the following linear search code:

```
int linear_search(int key, int A[], int n) {
    int i;
    #pragma omp parallel for num_threads(thread_count)
    for (i = 0; i < n; i++)
        if (A[i] == key)
            return i;
    return -1; // key not found
}
```

- This produces an error:
error: invalid exit from OpenMP structured block
because return causes an **invalid exit** from the parallel region.
- A more serious issue occurs when **iterations depend on previous iterations**, known as **data dependence**.

Example – Fibonacci Sequence:

```
arduino

fibonacci[0] = fibonacci[1] = 1;
#pragma omp parallel for num_threads(thread_count)
for (i = 2; i < n; i++)
    fibonacci[i] = fibonacci[i-1] + fibonacci[i-2];
```

Although the compiler accepts it, the output becomes **unpredictable** when run with multiple threads.

This is because the computation of fibo[i] depends on the values of fibo[i-1] and fibo[i-2], which may not be computed yet by other threads.

For instance:

- One thread may compute fibo[2] to fibo[5]
 - Another may start from fibo[6] before the first thread finishes
→ leading to incorrect results such as 1 1 2 3 5 8 0 0 0 0.
1. OpenMP **does not automatically detect data dependences** — the programmer must ensure loops are independent.
 2. Loops where one iteration **depends on results from another** (called **loop-carried dependences**) **cannot be safely parallelized** using parallel for.
 3. Such loops may need **sequential execution** or advanced OpenMP features like the **Tasking API** for correct parallelization.

4.5.3 Finding Loop-Carried Dependences

When using the **parallel for** directive, we only need to check for **loop-carried dependences** — not all kinds of data dependences.

Example :

```
cpp

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

Here, there is a data dependence between $x[i]$ (line 2) and $y[i]$ (line 3), but since both refer to the **same iteration**, it is **not loop-carried**.

Hence, the parallel version is valid:

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

Each thread works on different values of i , so no conflicts occur.

To find **loop-carried dependences**, we focus on **variables that are updated (written)** inside the loop.

If a variable is **written in one iteration** and **read or written in another**, it creates a **loop-carried dependence**, preventing safe parallelization.

4.5.4 Estimating π

The formula for π is:

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

Serial version:

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

Here, factor must be a **double** to ensure floating-point accuracy during division.

If we directly parallelize this with OpenMP:

```
#pragma omp parallel for num_threads(thread_count) reduction(+:sum)
for (k = 0; k < n; k++) {
    sum += factor / (2 * k + 1);
    factor = -factor;
}
```

the output becomes **incorrect** because factor depends on its previous value — this creates a **loop-carried dependence**.

Different threads updating factor simultaneously cause data inconsistency. To fix

it, calculate factor independently for each iteration:

```
factor = (k % 2 == 0) ? 1.0 : -1.0;
sum += factor / (2 * k + 1);
```

However, factor is still **shared** among threads by default.

We must make it **private** to avoid interference between threads:

```
double sum = 0.0;
#pragma omp parallel for num_threads(thread_count) \
reduction(+:sum) private(factor)
for (k = 0; k < n; k++) {
    if (k % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;
    sum += factor / (2 * k + 1);
}
pi_approx = 4.0 * sum;
```

- `reduction(+:sum)` ensures each thread keeps a local sum and combines them safely.
- `private(factor)` ensures each thread has its own copy of factor.

Variables with private scope are uninitialized at the start and undefined after the parallel block.

```
int x = 5;
#pragma omp parallel num_threads(thread_count) private(x)
{
    int my_rank = omp_get_thread_num();
    printf("Thread %d > before init, x = %d\n", my_rank, x);
    x = 2 * my_rank + 2;
    printf("Thread %d > after init, x = %d\n", my_rank, x);
}
printf("After parallel block, x = %d\n", x);
```

4.5.5 More on Scope

In OpenMP, correctly specifying the **scope of variables** inside a parallel or parallel for block is crucial to ensure correct execution and avoid data conflicts.

The issue we saw with the variable factor while estimating π is quite common.

To handle such cases properly, programmers should **explicitly declare the scope** of all variables instead of relying on OpenMP's default behavior.

The default(none) Clause

By adding:

```
default(none)
```

to a parallel directive, OpenMP **forces the programmer** to explicitly specify the scope of every variable that is declared **outside** the parallel block.

- Variables declared **inside** the block are **automatically private** (each thread gets its own copy).
- Variables declared **outside** the block must be explicitly declared as shared, private, or reduction

Example – π Calculation

```
c
double sum = 0.0;
#pragma omp parallel for num_threads(thread_count) \
default(none) reduction(+:sum) private(k, factor) shared(n)
for (k = 0; k < n; k++) {
    if (k % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;
    sum += factor / (2 * k + 1);
}
```

In this code:

- sum → **reduction variable** (partially private, then combined)
- factor, k → **private** (each thread has its own copy)
- n → **shared** (same value for all threads)

4.6 More about loops in OpenMP: sorting

4.6.1 Bubble Sort

The **Bubble Sort** algorithm repeatedly compares adjacent elements and swaps them if they are in the wrong order. This process continues until the entire list is sorted in ascending order.

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

Here, a is an array of n integers.

The **outer loop** finds the largest element in each pass and places it at its correct position. After the first pass, the largest element is at $a[n-1]$; after the second, the next largest is at $a[n-2]$, and so on.

The **inner loop** performs pairwise comparisons and swaps elements to "bubble up" the largest value in the current pass.

However, **loop-carried dependencies** exist in both loops:

- The **outer loop** depends on results of previous iterations (each pass works on a partially sorted list from the prior pass).
- The **inner loop** also depends on results of earlier comparisons (a swap affects the next comparison).

Due to these dependencies, **Bubble Sort is difficult to parallelize** without rewriting the algorithm. The parallel for directive in OpenMP cannot be directly applied to these dependent loops.

4.6.2 Odd–Even Transposition Sort

Odd–even transposition sort is a **variation of bubble sort** that offers more opportunities for **parallelism**. It repeatedly compares and swaps adjacent elements, but alternates between **odd** and **even phases** to gradually bring the list into order.

The **serial version** of the algorithm is written as follows:

```
c
for (phase = 0; phase < n; phase++)
    if (phase % 2 == 0)
        for (i = 1; i < n; i += 2)
            if (a[i - 1] > a[i]) Swap(&a[i - 1], &a[i]);
    else
        for (i = 1; i < n - 1; i += 2)
            if (a[i] > a[i + 1]) Swap(&a[i], &a[i + 1]);
```

Here, *a* is a list of *n* integers sorted in **increasing order** after *n* phases.

During an **even phase** (*phase* % 2 == 0), the algorithm compares and swaps the pairs (*a*[0], *a*[1]), (*a*[2], *a*[3]), and so on.

During an **odd phase**, it compares and swaps (*a*[1], *a*[2]), (*a*[3], *a*[4]), etc. After all phases are completed, the list becomes sorted.

For example, if the initial list is *a* = {9, 7, 8, 6}, the algorithm performs a sequence of phases until the list becomes {6, 7, 8, 9}.

It can be observed that the **outer loop** has a **loop-carried dependence**, since each phase depends on the outcome of the previous phase. Therefore, the outer loop cannot be parallelized.

However, the **inner loops** within each phase **do not have any loop-carried dependence**, since they operate on **disjoint pairs**. Hence, they can be safely parallelized.

In OpenMP, this can be achieved using the **parallel** and **for directives**.

The team of threads is **forked once** before the outer loop and reused for each phase.

thus avoiding the repeated cost of creating and joining threads.

Each phase still executes synchronously because OpenMP provides an **implicit barrier** at the end of every for directive, ensuring that no thread begins the next phase until all have completed the current one.

This improved OpenMP implementation is shown in **Program 5.5**, where the threads are created once using a parallel directive and the for directive is used within the block to parallelize the inner loops.

```

1 # pragma omp parallel num_threads(thread_count) \
2   default(none) shared(a, n) private(i, tmp, phase)
3   for (phase = 0; phase < n; phase++) {
4     if (phase % 2 == 0)
5     # pragma omp for
6       for (i = 1; i < n; i += 2) {
7         if (a[i-1] > a[i]) {
8           tmp = a[i-1];
9           a[i-1] = a[i];
10          a[i] = tmp;
11        }
12      }
13    else
14    # pragma omp for
15      for (i = 1; i < n-1; i += 2) {
16        if (a[i] > a[i+1]) {
17          tmp = a[i+1];
18          a[i+1] = a[i];
19          a[i] = tmp;
20        }
21      }
22  }

```

Program 5.5: Second OpenMP implementation of odd-even sort.

Performance results (Table 5.3) show that this version achieves better speedup, being about **17% faster** than the version that uses two parallel for directives, due to reduced thread management overhead.

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

4.7 Scheduling loops

When using parallel for, the assignment of loop iterations to threads is **system dependent**. Most OpenMP implementations use **block partitioning**, where:

- If there are n iterations and t threads,
 - Thread 0 gets $0 \rightarrow n/t - 1$

- Thread 1 gets $n/t \rightarrow 2n/t - 1$
- ...and so on.

Problem: If the work per iteration is **uneven**, some threads may do much more work than others.

```
sum = 0.0;
for (i = 0; i <= n; i++)
    sum += f(i); // f(i) takes more time as i increases
```

Using **block partitioning**, thread $t-1$ gets heavier iterations $\rightarrow$ poor load balancing. Using **cyclic partitioning**, iterations are assigned in a **round-robin** manner:

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

- Example program with $f(i)$ increasing in time:
 - Block assignment (2 threads) $\rightarrow$ speedup = 1.33
 - Cyclic assignment (2 threads) $\rightarrow$ speedup = 1.99

OpenMP solution: Use the **schedule clause** in parallel for or for directives to control iteration assignment and improve performance.

4.7.1 The Schedule Clause

The **schedule clause** controls how loop iterations are assigned to threads.

Default schedule:

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

Cyclic schedule (round-robin):

```
c
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);
```

Syntax:

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

Schedule types:

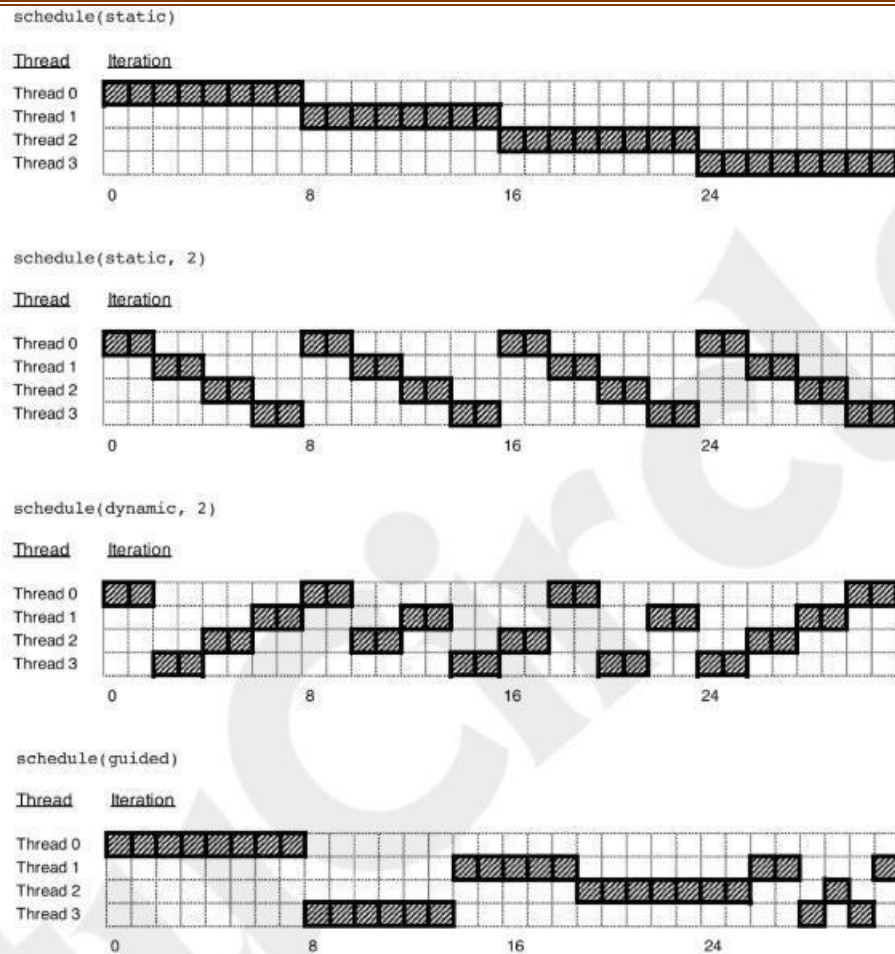
- **static:** Iterations assigned to threads before loop execution (can specify chunk size).
- **dynamic / guided:** Iterations assigned while the loop executes; threads request more work after finishing their current chunk.
- **auto:** Compiler/runtime decides.
- **runtime:** Determined at runtime via environment variable.

Chunk size:

- A block of consecutive iterations.
- Only applies to **static, dynamic, and guided** schedules.
- Determines how iterations are grouped before assignment to threads.

Visualization

- Shows static, dynamic, and guided scheduling and how work is divided among threads.

**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.

-

4.7.2 The Static Schedule type

In a **static schedule**, iterations are divided into **chunks** and assigned to threads **before the loop starts**, often in a **round-robin** fashion.

Examples: Suppose 12 iterations (0–11) and 3 threads:

- `schedule(static, 1)` – each thread gets 1 iteration at a time in round-robin:

```
yaml

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

- `schedule(static, 2)` – 2 iterations per chunk:

```
yaml

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

- `schedule(static, 4)` – 4 iterations per chunk:

```
yaml

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

- Most OpenMP implementations: `schedule(static, total_iterations / thread_count)`
- If chunksize is omitted, it's roughly `total_iterations / thread_count`.
- Iterations take roughly **equal time**.
- Can improve **memory access speed** on NUMA systems by keeping threads assigned to the same ranges across loops.

4.7.3 The *dynamic* and *guided Schedule* types

Dynamic Schedule

- Iterations are divided into **chunks** of consecutive iterations.
- **Threads execute a chunk** and request another from the run-time system when done.
- If chunksize is omitted, the default is **1**.
- **Advantages:** Useful when **iteration times vary**, since threads get work on a first-come, first-served basis.
- **Disadvantage:** Slight overhead due to dynamic allocation of chunks.
- **Tip:** Increasing chunksize reduces overhead but may reduce load balancing.

Guided Schedule

- Similar to **dynamic**, but **chunk sizes decrease** over time.
- The first chunk is large; subsequent chunks shrink roughly as:

$$\text{new_chunk_size} \approx \text{remaining_iterations} / \text{thread_count}$$

- If chunksize is not specified, chunks shrink to 1; if specified, they shrink to the given minimum.
- **Advantages:** Better load balancing for loops where later iterations are more compute-intensive.
- **Example:** Trapezoidal rule with $n = 10,000$ and 2 threads: first chunk ≈ 5000 , next ≈ 2500 , and so on.
- Table 5.4. We see that the size of the chunk is approximately the number of iterations remaining divided by the number of threads.

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

4.7.4 The runtime Schedule type

- Allows the **loop scheduling type** to be determined **at run-time** instead of at compile-time.
- Achieved using the clause:

```
#pragma omp parallel for schedule(runtime)
```

- The **environment variable** **OMP_SCHEDULE** controls the schedule type and chunk size.
- Values for **OMP_SCHEDULE** can be any valid OpenMP schedule: static, dynamic, guided, optionally with a chunksize.

Example:

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

This sets a **cyclic assignment** (iterations assigned one at a time to threads).

Advantages:

- Allows testing different schedules **without recompiling** the program.
- Useful for tuning performance, especially when loop iteration times are unpredictable.

Example usage in a bash script:

```
#!/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 [[ "${chunk_size}" != " " ]]; then
            sched_param="${schedule},${chunk_size}"
        fi
        OMP_SCHEDULE="${sched_param}" ./omp_mat_vect 4 2500 2500
    done
done
echo
done
```

4.7.5 Which Schedule?

Choosing a Schedule in OpenMP

- The **schedule clause** affects how loop iterations are divided among threads.
- Each schedule type introduces **different overheads**:
 - **Static**: Least overhead
 - **Dynamic**: More overhead (assigned at run-time)
 - **Guided**: Highest overhead (adaptive chunking)

Condition	Recommended Schedule	Reason
Each iteration takes similar time	Default or static	Minimal overhead, best performance
Iteration cost increases/decreases gradually	static, small chunksize (e.g., <code>schedule(static,1)</code>)	Balances workload smoothly
Iteration cost varies unpredictably	dynamic or guided	Allows flexible load balancing at run-time
Want to test different scheduling strategies	runtime	Can change behavior via OMP_SCHEDULE variable

Performance Tip

- Start with the **default schedule**.
- If performance is poor, **experiment** with static, dynamic, or guided schedules and different chunk sizes.
- Use `schedule(runtime)` for **easy tuning without recompiling**.

Example Observation

```
schedule(static,1)
```

Changing from the default to in a 2-thread, 10,000-iteration loop improved speedup from **1.33** → **1.99** (near optimal).

4.8 Producers and consumers

4.8.1 Queues

A **queue** is a linear data structure where:

- **Insertion** happens at the **rear**, and
- **Deletion** happens at the **front**.

It follows the **FIFO (First In, First Out)** principle — like customers waiting in line at a supermarket: new customers join at the end, and service starts from the front.

Key operations:

- **Enqueue:** Add an element to the rear.
- **Dequeue:** Remove an element from the front.

Applications in Computer Science:

- **Process scheduling:** Ensures fair access to resources (e.g., disk access).
- **Multithreaded systems:** Used for synchronization between
 - **Producer threads** (which generate tasks or requests) and
 - **Consumer threads** (which process those tasks).

Example:

Producers enqueue data requests (like stock prices), and consumers dequeue and process them — maintaining order and coordination between multiple threads.

4.8.2 Message Passing using Queues

In a **shared-memory system**, message passing between threads can be efficiently managed using **message queues**.

Each thread maintains its own **message queue**:

- To **send a message**, a thread **enqueues** a message into another thread's queue.
- To **receive a message**, a thread **dequeues** from its own queue.

This allows threads to **communicate and coordinate** safely without direct data sharing.

Program Behavior:

- Each thread repeatedly performs two actions:
 1. **Send_msg()** → generates a random integer message and a random destination thread, then enqueues the message.
 2. **Try_receive()** → checks its own queue; if a message is available, dequeues and prints it.
- Each thread sends a fixed number of messages (send_max).
- After sending all messages, it continues to receive until all threads finish.

```
for (sent_msgs = 0; sent_msgs < send_max; sent_msgs++) {
    Send_msg();      // Enqueue a random message to another thread
    Try_receive();   // Check and dequeue any received message
}
while (!Done())
    Try_receive();  // Keep receiving until all threads complete
```

4.8.3 Sending Messages

- When implementing message passing using queues, **sending (enqueueing)** a message becomes a **critical section**.
- This is because:
 - The message queue usually maintains a **rear pointer** (tail) to efficiently add new messages.
 - If **multiple threads** attempt to enqueue at the same time, the rear pointer may be **corrupted or overwritten**, leading to **message loss or inconsistency**.
 - Hence, enqueue operations must be **protected** to ensure mutual exclusion.
- **Critical Section:**
A code block where **shared data** (like the queue's rear pointer) is accessed or modified by multiple threads and therefore must be executed **by only one thread at a time**.

```
mesg = random();          // Generate random message
dest = random() % thread_count; // Select random destination thread

#pragma omp critical      // Protect enqueue operation
Enqueue(queue, dest, my_rank, mesg); // Add message to destination queue
```

- The #pragma omp critical directive ensures **safe concurrent access**.

- A thread can even send a **message to itself**, since all threads maintain their own message queue.
- Proper synchronization prevents **data corruption** and ensures **message integrity** in parallel environments.

4.8.4 Receiving Messages

Receiving (dequeuing) messages has **different synchronization requirements** from sending.

Each thread **owns its own queue**, meaning:

- **Only the owner thread** will dequeue messages.
- **Multiple threads** may enqueue messages to that queue.

Since dequeue is done only by one thread, **conflicts are minimal**.

If we track the number of **enqueued** and **dequeued** messages separately, we can compute the **queue size** as:

```
queue_size = enqueued - dequeued
```

- Only the owner thread updates dequeued.
- Multiple threads may update enqueued.

If two threads access enqueued simultaneously (one reading, one writing), the queue size might be off by ± 1 , but this only causes a **minor delay**, not data corruption.

Synchronization Strategy:

- No synchronization needed when queue has **2 or more messages**.
- Use a **critical section** only when there's **exactly 1 message** — to avoid possible race conditions between enqueue and dequeue.

```
queue_size = enqueued - dequeued;

if (queue_size == 0)
    return;                                // No messages to receive
else if (queue_size == 1)
    #pragma omp critical
    Dequeue(queue, &src, &msg);           // Safely dequeue the only message
else
    Dequeue(queue, &src, &msg);           // Safe: multiple messages exist

Print_message(src, msg);                   // Display received message
```

4.8.5 Termination Detection

In a message-passing program, threads must know **when to stop** receiving messages. This is handled by the **Done()** function.

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

is **incorrect**, because:

- Another thread may **send a message** immediately **after** this check.
- The receiving thread might **terminate too early**, leaving some messages undelivered.

Improved Approach:

After all threads finish sending messages, no new messages will be enqueued.

So, introduce a global counter **done_sending**, incremented by each thread when it finishes sending.

```
queue_size = enqueued - dequeued;

if (queue_size == 0 && done_sending == thread_count)
    return TRUE;      // No pending messages, and all threads finished sending
else
    return FALSE;
```

4.8.6 Startup

When the program starts, a **single master thread** begins execution. It reads the command-line arguments and **allocates an array of message queues**, one for each thread. This array must be **shared among all threads** so that any thread can send messages to any other thread.

Each **message queue** (implemented as a struct) contains:

- a list of messages
- front and rear pointers (or indices)
- count of messages enqueued
- count of messages dequeued

To reduce copying, the array should hold **pointers to these structs**. Once the master thread allocates the array, it starts all threads using a **parallel directive**, and each thread allocates memory for its own queue.

However, a problem can occur if some threads finish allocation earlier and try to enqueue messages into a queue that hasn't been allocated yet—this can cause a

program crash. To prevent this, all threads must wait until every queue is allocated before sending messages.

OpenMP provides a way to synchronize threads using the **barrier directive**:

```
#pragma omp barrier
```

This ensures that **no thread proceeds** beyond the barrier until **all threads** have reached it, guaranteeing safe and synchronized startup.

4.8.7 The *atomic* directive

After completing its message sends, each thread increments the variable **done_sending** before entering its final loop of receives. Since multiple threads update this variable, it forms a **critical section** that must be protected from concurrent access.

Although we could use the **critical** directive, OpenMP provides a more efficient option — the **atomic** directive:

```
#pragma omp atomic
```

The **atomic directive** ensures that a single **assignment operation** on a shared variable is executed **atomically** — that is, **without interruption** by other threads. However, it can only protect **simple update statements** such as:

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

where <op> can be one of +, -, *, /, &, ^, |, <<, or >>.

Note that <expression> must **not reference x**, and only the **load and store of x** are guaranteed to be atomic. For example:

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

Here, the update to x is atomic, but the increment of y is **not protected**, possibly leading to unpredictable results.

The **purpose of atomic** is performance: many processors support a **hardware-level load–modify–store instruction**, which allows such simple updates to be done safely and much faster than a general critical section.

4.8.8 Critical sections and locks

In the message-passing program, OpenMP’s **critical directive** is used to ensure that only one thread executes a block of code at a time. However, this can affect performance if used carelessly.

In earlier examples, we had just one critical section, so mutual exclusion was simple. But in this program, there are three such operations:

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

We don’t need complete mutual exclusion across all these—different threads can safely enqueue messages in different queues at the same time. But OpenMP treats **all unnamed critical sections as one composite section**, which **serializes** all such operations and **reduces parallelism**.

To improve this, OpenMP allows **named critical sections**:

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

Blocks with **different names** can execute simultaneously. However, names are fixed at **compile time**, so we can’t create a unique critical section for each thread dynamically. Hence, we use **locks** instead.

A **lock** is a shared data structure that allows programmers to manually control access to critical sections. Its usage follows this pattern:

```
// Executed by one thread
Initialize the lock;

// Executed by multiple threads
Set or acquire the lock;
Critical section;
Unset or release the lock;

// Executed by one thread
Destroy the lock;
```

When a thread calls the **lock function**, it either acquires the lock (if available) or waits until it is released. Once inside the critical section, no other thread can enter until the lock is **unset**.

OpenMP provides two kinds of locks:

- **Simple locks** – can be set once before being unset.
- **Nested locks** – can be set multiple times by the same thread.

For our program, **simple locks** are sufficient. The functions are declared in `omp.h`:

```
void omp_init_lock(omp_lock_t *lock_p);    // Initialize the Lock
void omp_set_lock(omp_lock_t *lock_p);    // Acquire the Lock (blocks if in use)
void omp_unset_lock(omp_lock_t *lock_p);  // Release the Lock
void omp_destroy_lock(omp_lock_t *lock_p); // Destroy the Lock
```

Thus, by using locks instead of generic critical sections, we can ensure **safe and efficient synchronization** while preserving **maximum parallel performance**.

4.8.9 Using locks in the message-passing program

In the message-passing program, we need **mutual exclusion** for each **individual message queue**, not for a single block of code shared by all threads. Using **locks** allows this finer control.

By adding a data member of type **omp_lock_t** to the **queue struct**, each queue maintains its own lock. Whenever a thread accesses a queue, it first sets the lock to ensure exclusive access, performs the operation, and then unsets the lock.

For example, the earlier code using a *critical* section:

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

is replaced with:

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

Similarly, for receiving messages:

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

Now, **only threads accessing the same queue** block each other, allowing true parallel execution. In contrast, the earlier critical directive forced **all threads** to wait, even if they were accessing different queues.

It's also possible to place the lock operations inside the Enqueue and Dequeue functions themselves. However, this could reduce performance, especially if Dequeue is frequently called by Try_receive. To maintain the current program structure and efficiency, it's better to keep the omp_set_lock and omp_unset_lock calls in the **Send** and **Try_receive** functions.

Since each queue now contains its own lock, it should be:

initialized when the queue is created (in the queue initialization function), using

```
omp_init_lock(&q_p->lock);
```

and **destroyed** by the owning thread before freeing the queue, using

```
omp_destroy_lock(&q_p->lock);
```

This ensures safe, fine-grained synchronization and better parallel performance in the message-passing system.

4.8.10 Critical directives, atomic directives, or locks?

When deciding between **critical directives, atomic directives, and locks** in OpenMP, the choice depends on the **type of mutual exclusion required** and **performance considerations**.

1. Atomic directive

- Typically the **fastest option** for simple updates.
- Works only for single assignment or increment/decrement statements of the forms allowed by OpenMP.
- Guarantees atomicity for that statement, but may enforce mutual exclusion **across all atomic directives** in some implementations.
- Limitation: If multiple atomic directives protect **different variables**, one might still block the other in some implementations, though standard does not require it.

2. Critical directive

- Simple and easy to use for any block of code.
- **Unnamed critical sections** are treated as a single composite section; only one thread executes any unnamed critical section at a time.
- **Named critical sections** allow different sections to execute concurrently.
- Performance is generally comparable to locks in most OpenMP implementations.

3. Locks

- Provide **fine-grained control** over mutual exclusion, especially useful for **data structures**, such as individual queues, rather than general code blocks.
- Can be initialized, set, unset, and destroyed explicitly, allowing concurrent access to different resources safely.
- Slightly more complex to use than critical directives but essential when **per-resource synchronization** is required.

4.8.11 Some caveats

When using **mutual exclusion** in OpenMP, there are several important **caveats** to keep in mind:

1. Don't mix atomic and critical on the same variable

Example:

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

Here, the atomic directive does not protect the second operation, so the two blocks **can interfere**, producing incorrect results.

Solution: either rewrite the code so atomic can be used for both, or protect both with a **critical directive**.

2. No guarantee of fairness

Threads waiting to enter a critical section may be **blocked indefinitely** if other threads repeatedly acquire it.

Example:

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

Thread 1 could potentially never execute the assignment if other threads keep entering the critical section first.

3. Avoid nesting critical sections

Example of **deadlock**:

```
#pragma omp critical
y = f(x);

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

If a thread inside the first critical section tries to enter the second, it **blocks forever**, causing a deadlock.

Using **named critical sections** can sometimes solve this:

```
#pragma omp critical(one)
y = f(x);

double f(double x) {
    #pragma omp critical(two)
    z = g(x);
}
```

However, deadlocks can still occur if threads enter critical sections in **different orders**. For example:

Time	Thread u	Thread v
0	Enter one	Enter two
1	Attempt two	Attempt one

Both threads remain **blocked forever**.

4.9 Caches, Cache Coherence, and False Sharing

1. Introduction to Cache Memory

- Over the years, the **speed of processors** has increased much faster than the speed of **main memory**.
- When a processor accesses data directly from main memory for each operation, most of its time is wasted waiting for the data to arrive.
- To overcome this, modern processors use **cache memory**, which is a small but much faster memory located close to the processor.
- Cache design is based on the principles of **temporal locality** (recently used data is likely to be reused soon) and **spatial locality** (data near recently accessed locations is likely to be used soon).
- Instead of transferring just one memory location, a **block of memory** (called a *cache line* or *cache block*) is transferred between the processor's cache and main memory.

2. Cache and Shared Memory

- In shared-memory systems, caches can significantly affect performance.
- Suppose a shared variable $x = 5$ is accessed by both thread 0 and thread 1.
 - Each thread loads x into its own cache and executes statements like $\text{my_y} = x$.
 - If thread 0 then executes $x++$, and thread 1 later executes $\text{my_z} = x$, what value will my_z hold—5 or 6?
- The issue arises because there are **multiple copies** of x :
 - one in **main memory**,
 - one in **thread 0's cache**, and
 - one in **thread 1's cache**.
- When thread 0 updates x , the other copies may become outdated.
 - This is known as the **cache coherence problem**.
- To solve this, most systems enforce **cache coherence** by:
 - marking cache lines invalid when data is modified elsewhere, and
 - ensuring updated data is fetched before further access.
- Thus, caches must communicate to stay synchronized.
 - For more details, refer to **Table 5.5** (sequence of accesses)

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	???	—	$x = 6$	$\text{my_z} = x$	???

3. Illustration: Matrix–Vector Multiplication

- Consider a matrix $A = (a_{ij})$ of size $m \times n$ and a vector x with n components.
 - Their product $y = Ax$ is a vector with m components.
 - Each element is computed as:

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

- Refer to **Figure 5.6** for a schematic view.

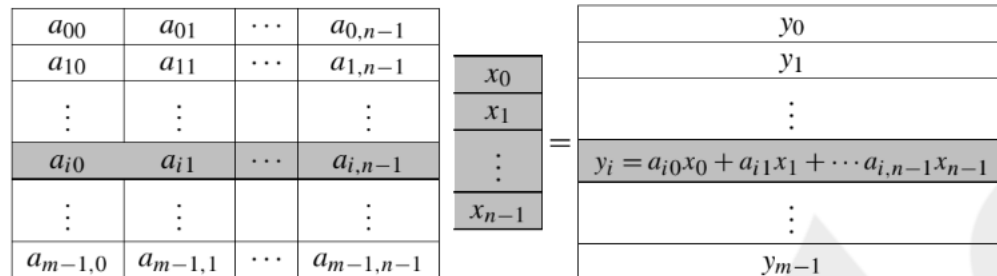


FIGURE 5.6

Matrix-vector multiplication.

- The **serial implementation** is as follows:

```
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 dependencies** in the outer loop because:
 - A and x are read-only, and
 - each iteration updates a separate element of y .
- Hence, we can **parallelize** the outer loop using OpenMP:
- #pragma omp parallel for num_threads(thread_count) \
- default(none) private(i,j) shared(A,x,y,m,n)

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

4. Performance Metrics

- If T_{serial} is the run time of the serial version and $T_{parallel}$ is the parallel version run time, then **Efficiency (E)** is given by:

$$E = \frac{S}{t} = \frac{T_{serial}}{t \times T_{parallel}}$$

where S is the speedup and t is the number of threads.

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

Threads	Matrix Dimension					
	8,000,000 × 8		8000 × 8000		8 × 8,000,000	
	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

- Table 5.6** lists the run times and efficiencies for various input sizes.
 - All cases involve the same total number of floating-point operations (64 million).
 - However, execution times differ due to cache performance.

5. Cache Misses and Performance Variation

- Write-miss:** occurs when a processor writes to a variable that is not in cache, requiring access to main memory.
- Read-miss:** occurs when a processor reads a variable that is not in cache.
- Using a cache profiler such as **Valgrind**, the following was observed:
 - For the **8,000,000 × 8** input, there were more *write-misses*—mostly due to initializing the larger y array.
 - For the **8 × 8,000,000** input, there were more *read-misses*—mostly due to frequent access of the larger x array.
- Hence, differences in cache performance significantly affect the execution time even when arithmetic operations are constant.

6. Parallel Efficiency and Cache Behavior

- As the number of threads increases, efficiency decreases—especially for the **8 × 8,000,000** case.
- For two threads, efficiency drops by over **20%**, and for four threads, by **50%**, compared to the other two cases.
- The reason again lies in **cache behavior** and particularly **cache coherence**

7. Understanding False Sharing

- **Cache coherence is maintained at the cache-line level**, typically 64 bytes (i.e., 8 doubles).
- When one core modifies *any variable* in a cache line, all other cores' copies of that line are invalidated—even if they use different variables in the same line.
- This leads to a phenomenon called **false sharing**.

Example:

- In the $8 \times 8,000,000$ problem with four threads:
 - If all y elements lie within one cache line, each thread's update to its assigned y element invalidates others' cache lines.
 - Each thread must repeatedly reload data from main memory, drastically slowing execution.
 - This occurs even though threads access **different variables**, not shared ones.
- Therefore, false sharing occurs when:
 - Threads update **different variables located in the same cache line**, and
 - Cache controllers treat the entire line as modified and invalidate others' copies.

8. Why False Sharing Does Not Always Occur

- For the 8000×8000 case:
 - Each thread processes a distinct range of y values, e.g.
 - Thread 2 $\rightarrow y[4000]$ to $y[5999]$
 - Thread 3 $\rightarrow y[6000]$ to $y[7999]$
 - Only at the interface (between $y[5999]$ and $y[6000]$) might a single cache line be shared.
 - However, accesses occur at different times, minimizing interference.
- Hence, false sharing has **negligible impact** on this case and the $8,000,000 \times 8$ case.
- Arrays A and x are read-only, so **no cache updates** occur there.

9. Techniques to Avoid False Sharing

- Two common approaches to prevent false sharing:
 1. **Padding** – Insert dummy elements in y to ensure each thread's data falls into different cache lines.
 2. **Private Buffers** – Let each thread use its private copy of y during computation and then update the shared array afterward.

4.10 Tasking in OpenMP

Traditional OpenMP constructs such as *parallel* and *parallel for* work efficiently when the number of iterations or parallel blocks is fixed and known. However, several real-world problems—like **web servers**, **recursive algorithms**, and **producer–consumer programs**—have dynamic or unpredictable workloads that cannot be parallelized using these constructs, since *while* and *do–while* loops or unbounded *for* loops are not supported. To address this, **OpenMP 3.0 introduced Tasking functionality**, enabling the creation of independent computation units called **tasks**, which the runtime schedules dynamically.

The **task directive** (`#pragma omp task`) defines a new unit of work. When a thread encounters it, the OpenMP runtime creates a task that may execute later depending on system load. Tasks must be generated **within a parallel region**, usually under a **single directive**, ensuring that only one thread launches them:

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

This structure prevents multiple threads from redundantly creating identical tasks.

A classic example is the **recursive Fibonacci computation**. In the serial version, recursive calls `fib(n-1)` and `fib(n-2)` are executed sequentially. By adding task directives before these calls, each can be executed in parallel. However, results initially appear incorrect because variables like *i* and *j* become **private** to each task. Declaring them as **shared** ensures all tasks reference the same memory locations:

```
#pragma omp task shared(i)
i = fib(n - 1);
#pragma omp task shared(j)
j = fib(n - 2);
```

Even then, execution order is undefined, and a thread may proceed before subtasks complete. To synchronize them, we use the **taskwait directive**, which acts as a barrier ensuring all child tasks finish before the parent continues. The corrected program (Program 5.6) now produces accurate results.

```

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.

Despite correctness, **performance may degrade** for large n due to **task creation overhead**. Each task requires its own data environment, increasing runtime costs. To mitigate this, OpenMP allows **conditional task creation** using the *if* clause—for example, `#pragma omp task if(n > 20)`—so tasks are created only for larger values of n . Another optimization eliminates redundant tasks by letting the parent execute one recursive call directly. On a 64-core system, these optimizations roughly halved the execution time for $n = 35$.

4.11 Thread-Safety

In shared-memory programming, **thread-safety** refers to the ability of a block of code or a function to be executed simultaneously by multiple threads **without causing errors or data inconsistencies**. A program is thread-safe if it ensures correct results even when multiple threads access shared resources concurrently.

For example, consider a program that uses multiple threads to **tokenize lines of a text file**. The function `strtok()` from `<string.h>` is commonly used for tokenization. Its first call takes a string to be tokenized, and subsequent calls with a NULL argument continue tokenizing the same cached string. Internally, `strtok()` maintains a **static variable** to remember the position in the string. When used by multiple threads, this cached pointer becomes **shared**, not private, leading to **incorrect results**—threads overwrite each other's tokenization data. Hence, `strtok()` is **not thread-safe**.

This problem is not unique to `strtok`; other C library functions such as `rand()` in `<stdlib.h>` and `localtime()` in `<time.h>` are also not guaranteed to be thread-safe. To overcome this issue, the C standard provides **re-entrant or thread-safe versions** of such functions. The thread-safe version of `strtok()` is `strtok_r()`, whose prototype is:

```
char *strtok_r(char *string, const char *separators, char **saveptr);
```

The third argument, `saveptr`, is a user-managed pointer that keeps track of the position in the string, replacing the internal static variable. Each thread can maintain its own `saveptr`, making the function **re-entrant** and **thread-safe**.

Thread-safety issues are often difficult to detect because incorrect programs may sometimes produce correct output due to the **non-deterministic scheduling of threads**. Such errors, known as **race conditions**, occur unpredictably and may appear only in certain runs. Therefore, ensuring thread-safety through **proper data scoping** and **use of re-entrant functions** is essential for reliable parallel programs.

MODULE 5

GPU programming with CUDA: GPUs and GPGPU, GPU architectures, Heterogeneous computing, Threads, blocks, and grids Nvidia compute capabilities and device architectures, Vector addition, returning results from CUDA kernels, CUDA trapezoidal rule I, CUDA trapezoidal rule II: improving performance, CUDA trapezoidal rule III: blocks with more than one warp

5.1 GPUs and GPGPU

In the late 1990s and early 2000s, the demand for highly realistic video games and animations drove the development of **powerful graphics processing units (GPUs)**. These processors were specifically designed to accelerate programs that rendered complex and detailed images. As the computational capabilities of GPUs became evident, programmers beyond the field of computer graphics began exploring their use for **general computational problems**, such as searching, sorting, and numerical simulations. This shift gave rise to **General-Purpose computing on GPUs (GPGPU)**.

Early attempts at GPGPU faced significant challenges because GPUs could only be programmed using **graphics APIs** like Direct3D and OpenGL. Programmers had to reformulate conventional algorithms in terms of graphics concepts such as **vertices, triangles, and pixels**, making development cumbersome and complex. To overcome this, researchers developed **languages and compilers** that allowed general algorithms to be implemented on GPUs using constructs similar to high-level CPU programming.

These developments eventually led to widely used APIs for general-purpose GPU programming. **CUDA**, developed by Nvidia, is optimized for Nvidia GPUs and requires relatively modest setup. **OpenCL**, in contrast, was designed for **portability**, allowing programs to run on a variety of devices, including GPUs, **FPGAs**, and **digital signal processors (DSPs)**, but it demands more detailed setup to handle different systems. Due to its simplicity for Nvidia GPUs, CUDA is often preferred in practice.

This evolution illustrates how GPUs, originally tailored for graphics, have become powerful tools for accelerating a broad range of **parallel computations**, forming a crucial part of modern high-performance computing.

5.1 GPUs architectures

A **CPU** is generally a **SISD (Single Instruction, Single Data)** device in **Flynn's taxonomy**, where each instruction operates on a small set of data. In contrast, **GPUs** are **SIMD (Single Instruction, Multiple Data)** processors that apply one instruction to many data elements simultaneously, making them highly efficient for parallel workloads.

A **SIMD processor** consists of a single **control unit** and multiple **datapaths**. During this process, some datapaths remain idle while others execute, as shown in **Table 6.1**.

Table 6.1 Execution of branch on a SIMD system.

Time	Datapaths with $x[i] \geq 0$	Datapaths with $x[i] < 0$
1	Test $x[i] \geq 0$	Test $x[i] \geq 0$
2	$x[i] += 1$	Idle
3	Idle	$x[i] -= 2$

Modern GPUs consist of several **Streaming Multiprocessors (SMs)**, each containing many **Streaming Processors (SPs)**—the actual datapaths. SMs operate **asynchronously**, allowing different branches of code to execute on different SMs without penalty, reducing execution stages as shown in **Table 6.2**. Nvidia GPUs, for instance, can have up to **82 SMs** with **128 SPs** each, totaling over **10,000 SPs**. Nvidia refers to this model as **SIMT (Single Instruction, Multiple Thread)**, since threads may execute at different times to hide memory latency.

Table 6.2 Execution of branch on a system with multiple SMs.

Time	Datapaths with $x[i] \geq 0$ (on SM A)	Datapaths with $x[i] < 0$ (on SM B)
1	Test $x[i] \geq 0$	Test $x[i] \geq 0$
2	$x[i] += 1$	$x[i] -= 2$

Each SM includes a small **shared memory** accessible quickly by its SPs, while all SMs share a larger, slower **global memory** (see **Figure 6.1**).

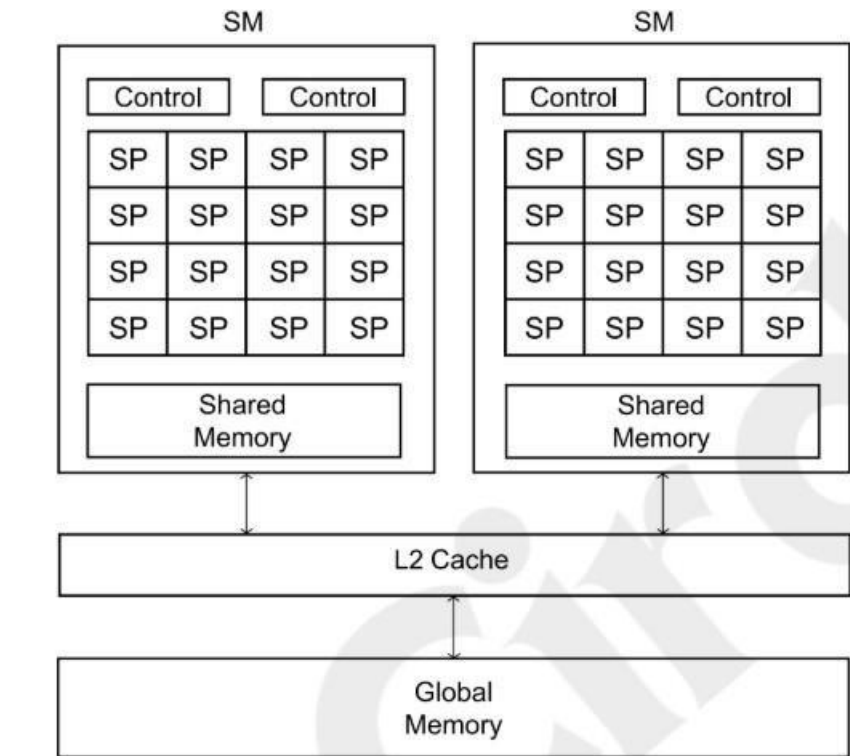


FIGURE 6.1
Simplified block diagram of a GPU.

The **CPU and GPU** have separate memories—the **host** and **device**, respectively. Earlier systems required explicit data transfers between them, but newer Nvidia architectures (with **compute capability ≥ 3.0**) support **unified memory**, eliminating the need for explicit transfers while retaining performance benefits (see **Figure 6.2**).

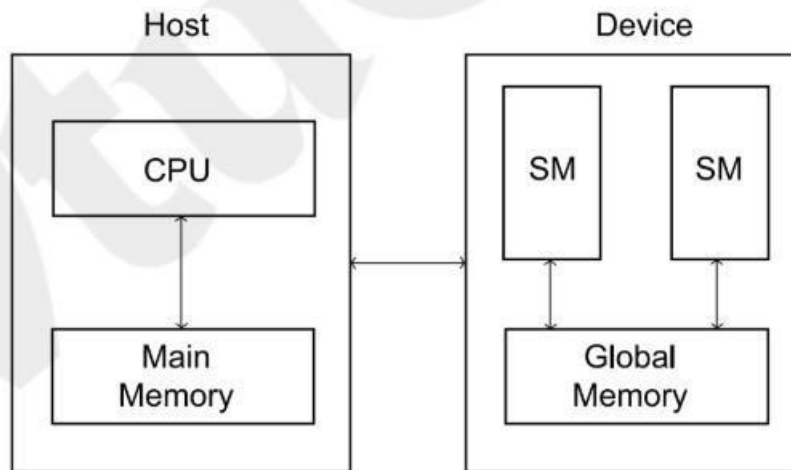


FIGURE 6.2

Simplified block diagram of a CPU and a GPU.

Thus, GPU architecture achieves massive **parallelism** and **high throughput** by combining SIMT execution, fast shared memory, and asynchronous multiprocessors, making it central to modern parallel computing.

5.3 Heterogeneous computing

Heterogeneous computing refers to systems that use processors with **different architectures** working together to execute a program. When we write programs that utilize both a **CPU (host processor)** and a **GPU (device processor)**, we are engaging in heterogeneous computing. Though we write a single program following the **SPMD (Single Program, Multiple Data)** model, parts of the program are designed for the CPU and other parts for the GPU, effectively creating two cooperating programs within one.

This approach has gained significant importance in recent years. Between 1986 and 2003, **CPU single-thread performance** grew by over 50% annually. However, after 2003, this growth slowed dramatically—to less than **4% per year** between 2015 and 2017. As a result, programmers began exploring other hardware to achieve better performance. **GPUs**, with their massive parallelism, are one major solution, but other specialized processors also play a role.

Field Programmable Gate Arrays (FPGAs) offer **programmable logic blocks and interconnects**, allowing reconfiguration before program execution to suit specific tasks. **Digital Signal Processors (DSPs)**, on the other hand, are designed with **special circuitry** for processing and transforming signals, such as compression or filtering of **real-world analog data**.

Thus, heterogeneous computing represents a powerful paradigm in modern systems, combining different processor types—each with its strengths—to achieve **higher performance and efficiency** than homogeneous CPU-only architectures.

5.4 CUDA hello

CUDA (Compute Unified Device Architecture) is a **software platform developed by NVIDIA** for programming heterogeneous systems that include both **CPU (host)** and **GPU (device)**. It enables developers to write **GPGPU (General-Purpose computing on Graphics Processing Units)** applications, allowing the GPU to be used for computations beyond traditional graphics processing.

Although CUDA originally stood for *Compute Unified Device Architecture*, NVIDIA now treats it simply as the name of their **API for GPU programming**.

CUDA provides **language-specific APIs** for several languages such as **C, C++, Fortran, Python, and Java**. In most engineering applications, we use **CUDA C**, which is an extension of the C programming language. However, since the CUDA compiler is a **modified C++ compiler**, certain C++ constructs may appear even in CUDA C programs. For instance, C requires no cast when using `malloc()`, while C++ does:

```
c

// C version
float *x = malloc(n * sizeof(float));

// C++ version
float *x = (float *) malloc(n * sizeof(float));
```

Host and Device Programs

A CUDA program is **heterogeneous**, meaning it contains:

- A **Host (CPU)** part – executed on the main processor.
- A **Device (GPU)** part – executed on the GPU cores.

When we write a CUDA program, we are effectively writing **two interacting programs**: one running on the CPU and one on the GPU. The CPU launches and manages tasks on the GPU using CUDA API calls.

Compilation

CUDA programs **cannot be compiled with a regular C compiler (like gcc)**. This is because the code must be compiled into **machine instructions for two architectures**—one for the CPU and another for the GPU.

To do this, we use **nvcc**, the NVIDIA CUDA compiler, which separates host and device code and compiles them appropriately.

Example: CUDA “Hello, World” Program

The simplest example in CUDA is similar in spirit to the classic “hello, world” program, except that **each CUDA thread prints a greeting** from the GPU. This demonstrates that GPU threads can execute code independently.

5.4.1 The source code

The CUDA program begins with including the **CUDA header file**. The **Hello** function, defined with the keyword **__global__**, is a **kernel**—a function started by the host (CPU) but executed by each thread on the **device (GPU)**. All CUDA kernels have a **void return type**.

The **main** function, like in ordinary C programs, runs on the host. It reads the number of threads from the command line and launches that many copies of the kernel using the syntax `Hello<<<1, n>>>()`. The call to **cudaDeviceSynchronize()** ensures that the CPU waits until all GPU threads have completed execution. Once all threads finish, the program ends normally with **return 0**.

```

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!\n", threadIdx.x);
8  } /* Hello */
9
10
11 /* Host code: Runs on CPU */
12 int main(int argc, char* argv[]) {
13     int thread_count;    /* Number of threads to run on GPU */
14
15     thread_count = strtol(argv[1], NULL, 10);
16                     /* Get thread_count from command line */
17
18     Hello <<<1, thread_count>>>();
19                     /* Start thread_count threads on GPU, */
20
21     cudaDeviceSynchronize();    /* Wait for GPU to finish */
22
23     return 0;
24 } /* main */

```

Program 6.1: CUDA program that prints greetings from the threads.

5.4.2 Compiling and running the program

A CUDA source file containing both host and device code must use the **“.cu”** extension. For example, the hello program is saved as **cuda_hello.cu**. It is compiled using the **NVIDIA CUDA compiler (nvcc)**, which generates code for both CPU and GPU.

To compile:

```

ruby
$ nvcc -o cuda_hello cuda_hello.cu

```

To run the program with **one GPU thread**:

```

shell
$ ./cuda_hello 1

```

Output:

```

cpp
Hello from thread 0!

```

To run the program with **ten GPU threads**:

```
shell
$ ./cuda_hello 10
```

Output:

```
Hello from thread 0!
Hello from thread 1!
Hello from thread 2!
```

```
...
```

```
Hello from thread 9!
```

Each line is printed by a **different GPU thread**, showing that multiple threads can execute the same kernel in parallel.

5.6 Threads, Blocks, and Grids in CUDA

In CUDA, **threads** are the smallest units of execution, and they run on the **Streaming Processors (SPs)** within a GPU's **Streaming Multiprocessors (SMs)**.

When we launch a kernel like

```
Hello<<<1, thread_count>>>();
```

the numbers inside the angle brackets specify how the work is divided. The **first value (1)** represents the **number of thread blocks**, and the **second (thread_count)** represents the **number of threads per block**.

An NVIDIA GPU consists of several **Streaming Multiprocessors (SMs)**, and each SM contains many **Streaming Processors (SPs)**. Every CUDA **thread** executes on one SP (

If the kernel is launched as

```
Hello<<<2, thread_count/2>>>();
```

and thread_count is even, CUDA creates two thread blocks, each with thread_count/2 threads. These blocks may be assigned to two different SMs.

CUDA organizes computation in a **hierarchical structure**

- **Threads** are grouped into **Blocks**, which execute on a single SM.
- **Blocks** are grouped into a **Grid**, which represents all threads launched by one kernel call.

Each thread can determine its position using predefined variables:

- threadIdx – thread's index within its block.
- blockIdx – block's index within the grid.

- gridDim – dimensions (number of blocks) in the grid.

These are 3D structures (x, y, and z) that allow mapping of threads to multi-dimensional data. For instance, in image or matrix applications, threadIdx.x and threadIdx.y can represent a pixel's row and column.

Example of a 3D grid and block configuration:

```
dim3 grid_dims(2, 3, 1);
dim3 block_dims(4, 4, 4);
Kernel<<<grid_dims, block_dims>>>();
```

This creates a grid of $2 \times 3 \times 1 = 6$ **blocks**, each with $4 \times 4 \times 4 = 64$ **threads**.

All thread blocks must have **identical dimensions** and be **independent**, meaning any block can finish execution regardless of others. This independence allows the GPU to schedule blocks in **any order or in parallel**, ensuring maximum flexibility and performance.

```
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.

5.7 NVIDIA Compute Capabilities and Device Architectures

Each NVIDIA GPU has specific hardware limits that determine how many **threads**, **blocks**, and **resources** it can handle. These limits are defined by what NVIDIA calls the **Compute Capability of the GPU**.

Department of CSD, ATMECE, Mysuru

The compute capability is written as a **version number a.b**, where

- **a** = major revision number (possible values: 1, 2, 3, 5, 6, 7, 8)
- **b** = minor revision number (ranges from 0–7 depending on the major version)

Devices with compute capability **below 3.0** are **no longer supported** by CUDA.

For GPUs with compute capability **greater than 1.x**:

- Maximum threads per block = **1024**
- For compute capability **2.b**, a single **Streaming Multiprocessor (SM)** can handle up to **1536 threads**
- For compute capability **greater than 2.x**, a single SM can manage up to **2048 threads**

There are also **dimension limits** for thread blocks and grids:

- For compute capability > 1:
 - Maximum block dimension in **x or y** = **1024**
 - Maximum block dimension in **z** = **64**

(Refer to *Table 6.3* for a list of NVIDIA GPU architectures and their corresponding compute capabilities.)

NVIDIA assigns **architecture names** (such as *Fermi*, *Kepler*, *Maxwell*, *Pascal*, *Volta*, *Turing*, *Ampere*, and *Ada Lovelace*) to its GPU microarchitectures. Note that *Tesla* is both a **GPU architecture name** and the **brand name** for NVIDIA's GPGPU product line, which can include standalone GPUs or **system-on-chip** devices.

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

Lastly, **CUDA API versions** evolve independently of **compute capabilities**, meaning a newer CUDA Toolkit can still support older GPUs, provided they meet the minimum compute capability requirement.

5.8 Vector addition

GPUs and CUDA are especially efficient when executing **data-parallel programs**, where the same operation is applied to many data elements simultaneously. A classic example is **vector addition**, which is also **embarrassingly parallel**—each computation is independent of the others.

We define three arrays of equal size n :

```
float *x, *y, *z;
```

The arrays x and y are initialized on the **host (CPU)**, and the **device (GPU)** performs the addition:

```
z[i] = x[i] + y[i];
```

Each GPU thread handles one element of the arrays, so a kernel will launch **at least n threads**, with thread *i* computing the *i*-th element of the result.

Since most GPUs are optimized for **32-bit (float)** operations rather than **64-bit (double)**, single-precision floats are preferred for performance.

After the kernel finishes execution, the host program checks the results, frees allocated memory, and exits.

```

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.

5.8.1 The Kernel – Vector Addition in CUDA

In the **kernel function** (Lines 1–11 of Program 6.3), each thread is responsible for computing one element of the output vector *z*. To do this, we must determine which element each thread should process.

CUDA threads are organized hierarchically into **blocks** and **grids**, so each thread's **global index (rank)** can be computed as:

```
rank = blockDim.x * blockIdx.x + threadIdx.x
```

This gives every thread a unique index across the entire grid.

For example, if there are **four blocks** with **five threads per block**, the total number of threads is $4 \times 5 = 20$, and the **global ranks** range from 0 to 19 (see *Table 6.4*).

Table 6.4 Global thread ranks or indexes in a grid with 4 blocks and 5 threads per block.

blockIdx.x	threadIdx.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

In the kernel, this index is stored in a variable such as `my_elt`, which is then used to access the elements of the arrays:

```
z[my_elt] = x[my_elt] + y[my_elt];
```

Before performing the addition, the program checks:

```
if (my_elt < n)
    z[my_elt] = x[my_elt] + y[my_elt];
```

This ensures that no thread tries to access elements beyond the array bounds when the total number of threads exceeds `n`.

For instance, if `n = 997`, we cannot evenly divide threads across blocks (since 997 is prime). We might choose **4 blocks of 256 threads** (total = 1024 threads), meaning the extra 27 threads simply skip the addition.

If CUDA support isn't available, the same logic can be implemented using a **serial vector addition** (as shown in *Program 6.4*).

```
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.

Thus, a **CUDA kernel** can be viewed as a **parallel version of a serial loop**, where each loop iteration is assigned to a different thread. Following **Foster's design methodology**, this program's tasks are independent additions—no communication or aggregation is needed, only a direct mapping of elements to threads.

In the main function, after declaring variables, the **Get_args** function is invoked to retrieve four parameters: **n** (the array size), **blk_ct** (number of thread blocks), **th_per_blk** (threads per block), and **i_g** (a flag indicating input mode). These values are obtained from the command line. If the arguments are missing or invalid, the function displays a usage message and stops execution. It also verifies that **n** does not exceed the total available threads; otherwise, it terminates with an error. When **i_g** specifies user input, arrays **x** and **y** are entered manually; otherwise, they are populated with random values. The **Get_args** function is written in standard C and executes entirely on the **host**. (See Program 6.5.)

```

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.

5.8.3 Allocate_vectors and managed memory

After obtaining the command-line parameters, the **main** function invokes **Allocate_vectors** to allocate four float arrays of size **n**: **x**, **y**, **z**, and **cz**. Among these, **x**, **y**, and **z** are used both on the host and device, while **cz** is used only on the host to compute the vector sum sequentially for verification of the device's result. (See Program 6.6.)

Since **cz** is host-only, its memory is allocated using the standard C library function **malloc()**. For **x**, **y**, and **z**, allocation is performed using the CUDA function **cudaMallocManaged()**, which provides *unified memory* accessible by both host and device:

```
_host_ cudaError_t cudaMallocManaged(void **devPtr, size_t size, unsigned flags);
```

The **_host_** qualifier indicates that the function executes on the host. The function returns a value of type **cudaError_t**, useful for error checking, though often omitted for brevity. The first argument is a pointer to the pointer being allocated, the second specifies the size in bytes, and the optional flags parameter controls kernel access (default: **cudaMemAttachGlobal**).

cudaMallocManaged() simplifies programming by providing a unified memory model where host and device share access to the same allocation, though it has certain requirements and trade-offs:

1. Requires devices with **compute capability ≥ 3.0** and a **64-bit host OS**.
2. For devices with **compute capability < 6.0** , unified memory cannot be accessed simultaneously by host and device.
3. Unified memory may be slower than explicit data transfers due to system-managed synchronization.

When using unified memory, the system automatically decides when to move data between host and device. While this eases programming, explicit memory management (via **cudaMalloc**, **cudaMemcpy**, etc.) can offer better performance through optimized or overlapped transfers.

5.8.4 Other functions called from main

Except for **Free_vectors**, all other host functions invoked from **main** are written in standard C. The **Init_vectors** function initializes arrays **x** and **y**—either by reading values from standard input using **scanf** or by generating them randomly using the **random()** library function, based on the command-line argument **i_g**.

The **Serial_vec_add** function (refer to *Program 6.4*) performs vector addition on the host using a simple for loop and stores the result in the array **cz**.

The **Two_norm_diff** function calculates the Euclidean distance between the vectors **z** (computed on the device) and **cz** (computed serially on the host). It computes the square of the difference for each corresponding pair of elements, sums them, and then takes the square root:

$$\sqrt{(z[0] - cz[0])^2 + (z[1] - cz[1])^2 + \dots + (z[n-1] - cz[n-1])^2}$$

```

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 releases all dynamically allocated memory. The host-only array **cz** is freed using the standard **free()** function, while the arrays **x**, **y**, and **z**, allocated via **cudaMallocManaged**, are freed using **cudaFree()**:

The **_device_** qualifier means this function can be invoked from either host or device code,
 Throughput of GPU, CPU, and Memory cannot be freed on the host and vice versa

It is crucial to explicitly free device memory within the program; otherwise, it remains allocated until program termination. This becomes important in applications invoking multiple kernels—if memory from earlier kernels isn't released, it stays occupied even if later kernels don't use it.

(Refer to 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.

5.8.5 Explicit memory transfers

In systems **without unified memory**, the vector addition program must explicitly handle memory transfers between the host and the device. The **kernel** itself remains unchanged—it receives arrays **x**, **y**, and **z**, along with **n**, computes the global thread index **my_elt**, and performs the addition if **my_elt < n**.

However, in the **main function**, since host and device pointers are distinct and not interchangeable, separate memory allocations are required. Thus, six arrays are declared—**hx**, **hy**, **hz** for host memory, and **dx**, **dy**, **dz** for device memory (Lines 15–16). Allocation is handled in the **Allocate_vectors** function (see *Program 6.10*). Instead of using **cudaMallocManaged**, the host arrays use the C **malloc()** function, while the device arrays use the CUDA **cudaMalloc()** function:

```
__host__ __device__ cudaError_t cudaMalloc(void **dev_p, size_t size);
```

Here, **dev_p** points to device memory, and **size** specifies the number of bytes to allocate.

Once the host arrays **hx** and **hy** are initialized, their data is transferred to device memory **dx** and **dy** using **cudaMemcpy()** (Lines 24–26):

```
__host__ __device__ cudaError_t cudaMemcpy(void *dest, const void *source, size_t count,
cudaMemcpyKind kind);
```

The **kind** argument specifies the transfer direction:

- **cudaMemcpyHostToDevice** → copy from host to device
- **cudaMemcpyDeviceToHost** → copy from device to host

The kernel launch (Line 28) uses **dx**, **dy**, and **dz**, which are valid device pointers. After execution, the results in **dz** are copied back to **hz** on the host (Line 31) using **cudaMemcpy()** again. Since **cudaMemcpy** is **synchronous**, it ensures that the kernel completes before data transfer begins, making an explicit **cudaDeviceSynchronize()** unnecessary.

Finally, the program verifies the results, releases memory, and exits. The **Free_vectors** function now frees both host and device allocations: **free()** for host memory and **cudaFree()** for device memory. Thus, the modified program manages **seven pointers** in total—three on the host, three on the device, and one for result verification.

```

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.

5.9 Returning results from CUDA kernels

CUDA kernels have several important restrictions and behaviors that programmers must keep in mind. **All kernels must have a return type of void**, meaning they cannot directly return values to the host. Similarly, they cannot use **standard C pass-by-reference** to send results back, since **host memory addresses are generally invalid on the device**, and vice versa.

For example, if we attempt to pass a host variable's address to a kernel (as in the code using &sum), the device will likely hang or the host will print the unchanged value because the device tried to write to an invalid address.

To "return" values from a kernel, several methods can be used:

1. **Using Unified Memory** – The host and device share memory allocated with `cudaMallocManaged`.

```

int *sum_p;
cudaMallocManaged(&sum_p, sizeof(int));
Add<<<1,1>>>(2, 3, sum_p);
cudaDeviceSynchronize();
printf("The sum is %d\n", *sum_p);
cudaFree(sum_p);

```

Here, the device writes to a managed pointer that is automatically synchronized with the host.

2. **Without Unified Memory** – Separate allocations are made for host and device, and data is explicitly copied back using `cudaMemcpy`.

```
int *hsum_p, *dsum_p;
hsum_p = (int*)malloc(sizeof(int));
cudaMalloc(&dsum_p, sizeof(int));
Add<<<1,1>>>(2, 3, dsum_p);
cudaMemcpy(hsum_p, dsum_p, sizeof(int), cudaMemcpyDeviceToHost);
printf("The sum is %d\n", *hsum_p);
```

This version ensures compatibility with older systems but requires manual data transfer.

3. Using a Managed Global Variable – A global variable can be declared with the qualifier `__managed__`, making it accessible from both the host and the device.

```
__managed__ int sum;
__global__ void Add(int x, int y) { sum = x + y; }
```

The host can then read `sum` after synchronization. However, this approach is only supported on devices with **compute capability** ≥ 3.0 , and **simultaneous host-device access** is restricted on devices below **6.0**.

Although the managed global variable method is convenient, it reduces **modularity** and is generally discouraged in complex programs. Thus, while CUDA provides flexible options for returning results from kernels, explicit memory management often remains the more robust and portable approach.

5.10 CUDA trapezoidal rule I

5.10.1 The trapezoidal rule

The text says “**Recall (see Sec. 3.2.1)**”, which refers to the earlier explanation of the trapezoidal rule in Chapter 3.

Figure 3.3 illustrates the **trapezoidal approximation of the area under a curve** — it visually shows dividing the interval $[a, b]$ into n subintervals and representing each part as a trapezoid.

Figure 3.4 (mentioned later) specifically shows **the i th trapezoid**, where h is the width and $f(x_i)$, $f(x_{i+1})$ are the two base lengths.

This figure visually corresponds to the expression

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

So when citing in your lecture notes or slides, you can keep the references like this:

The trapezoidal rule divides the interval $[a, b]$ into n equal subintervals (see Fig. 3.3), each approximated by a trapezoid of height $h = (b - a)/n$. For the i^{th} trapezoid (see Fig. 3.4), the area is $\frac{h}{2} [f(x_i) + f(x_{i+1})]$.

Then follow it with:

Hence, the total area is approximated as

$$\int_a^b f(x) dx \approx \frac{h}{2} \left[f(a) + f(b) + 2 \sum_{i=1}^{n-1} f(x_i) \right]$$

5.10.2 A CUDA implementation

In the serial version of the trapezoidal rule, most of the computation occurs within the **for loop** that evaluates the function $f(x_i)$ and accumulates the partial sums into the total area. According to **Foster's design methodology**, these two major tasks can be parallelized:

1. **Function evaluation** — computing $f(x_i)$ at each subinterval.
2. **Summation** — adding $f(x_i)$ to the partial sum, trap.

Since the second depends on the first, both can be combined into a single unit of work. Thus, in CUDA, each thread can be assigned one iteration of the serial loop — computing its own x_i and $f(x_i)$ and contributing to the total sum.

Each thread is identified by a unique **rank (thread index)**, similar to the vector addition program discussed earlier.

The basic mapping looks like this:

```
/* 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);
trap += my_trap;
```

However, this naive implementation introduces several **issues**:

1. **Uninitialized variables** — The parameters h and $trap$ must be properly initialized before use.
2. **Out-of-range thread indices** — The loop in the serial code runs from $i = 1$ to $i = n - 1$, but thread indices start from 0. Threads with indices outside this range must not execute computations.
3. **Race condition** — Multiple threads updating the shared variable $trap$ simultaneously can lead to incorrect results, since one thread may overwrite another's update. (Refer to **Section 2.4.3**, "Race Conditions.")
4. **Return type restriction** — CUDA kernels have a void return type, so results must be communicated through parameters or device memory.
5. **Final scaling** — After all threads complete their partial computations, the total sum must be multiplied by h to obtain the final area.

```

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.

Program 6.12 in the textbook presents a corrected CUDA implementation that resolves these issues — using shared memory, synchronization, and reduction techniques to safely combine partial sums from multiple threads.

5.10.3 Initialization, return value, and final update

To handle initialization and final computation (Items 1 and 5 from Section 6.10.2), one might consider assigning a **single thread**—for example, **thread 0 of block 0**—to perform these 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;

```

However, this approach introduces important **problems** related to **variable scope** and **thread synchronization**.

Private versus Shared Variables

In CUDA, each thread has its own **stack**, similar to threads in Pthreads and OpenMP. All **formal arguments** of a kernel are allocated on this per-thread stack, meaning that each thread gets its **own private copies** of variables such as `h` and `trap`. Hence, any modification to `h` or `trap` by one thread will not be visible to others.

The variable `h` can be safely initialized **once in the host** before launching the kernel, as its value remains constant for all threads. But `trap`, which accumulates partial results, must be **shared among threads** to correctly compute the total area.

Using Managed Memory for Shared Access

To achieve shared access, we allocate `trap` in **unified memory** (accessible by both host and device) and pass a **pointer** to it as a kernel argument:

Here, each thread receives its **own copy of the pointer** `trap_p`, but all these pointers reference the **same memory location**. Thus, the dereferenced variable `*trap_p` becomes **shared** among all threads.

This pointer-based approach also naturally resolves the earlier limitation that **CUDA kernels cannot return values** (Item 4 from Section 6.10.2), since results can now be written directly to the shared memory location.

Wrapper Function

As seen in **Program 6.12**, the CUDA trapezoidal rule employs a **wrapper function** named `Trap_wrapper`.

A **wrapper function** primarily exists to call another function—in this case, the kernel—while performing necessary setup and cleanup tasks.

It handles all **preparation before launching the kernel** (such as memory allocation, initialization, and parameter setup) and any **post-processing** (like scaling or memory deallocation) once the kernel completes execution.

5.10.4 Using the Correct Threads

In the CUDA implementation of the trapezoidal rule, we assume that the **total number of threads**, given by `blk_ct * th_per_blk`, is **at least as large as the number of trapezoids** `n`.

Recall that in the serial version, the for loop iterates over indices `i = 1, 2, ..., n - 1`.

This means that **thread 0** (corresponding to `i = 0`) and any **thread with index greater than or equal to `n`** should **not execute** the main computation portion of the kernel.

To enforce this, a simple conditional check is added before performing any calculations inside the kernel:

```

if (0 < my_i && my_i < n) {
    /* Compute x, f(x), and add the result to *trap_p */
    ...
}

```

This ensures that only threads corresponding to valid trapezoid indices participate in evaluating the function and updating the shared variable.

Such a check prevents **out-of-range accesses** and avoids unnecessary or invalid computations by extra threads that may have been launched to fill a complete block.

This conditional test appears as **Line 11 in Program 6.12**, where it forms an essential safeguard for correct parallel execution.

5.10.5 Updating the Return Value and atomicAdd

The next issue concerns updating the shared memory location `*trap_p` (Item 3 from the earlier list). Since this variable is **shared among threads**, a simple update like

```
*trap_p += my_trap;
```

creates a **race condition**. Multiple threads may attempt to modify `*trap_p` simultaneously, and because the operation is not atomic, the final value stored could be **unpredictable or incorrect**.

To eliminate this problem, CUDA provides **atomic operations**, which ensure that certain updates appear **indivisible** to all other threads. An operation is said to be *atomic* if, from the viewpoint of every thread, it happens all at once — no other thread can observe the intermediate state of the operation.

Thus, if one thread performs an atomic addition, any other thread's attempt to read or write to the same memory location will occur **entirely before or after** the operation — never during it. This guarantees consistent and correct results even in concurrent execution.

Addition, by default, is **not atomic**; it involves multiple machine instructions. Hence, CUDA provides special built-in atomic functions to perform such updates safely. The one relevant to our case is `atomicAdd`, defined as:

```

__device__ float atomicAdd(
    float* float_p, // in/out: address of variable in device memory
    float val       // in: value to add
);

```

This function **atomically adds** `val` to the value stored at the location pointed to by `float_p`, and then stores the result back in the same location.

It also returns the value that was originally present in `*float_p` at the start of the call.

In the CUDA trapezoidal rule implementation (Program 6.12), `atomicAdd` is used (see **Line 14**) to safely accumulate each thread's contribution (`my_trap`) to the shared total area variable `*trap_p`, ensuring correctness even when thousands of threads execute concurrently.

5.10.6 Performance of the CUDA trapezoidal rule

To determine the **run-time performance** of the CUDA trapezoidal rule, we measure the execution time of the **Trap_wrapper** function. This wrapper function encapsulates the entire trapezoidal rule computation — including initialization, kernel execution, and final updates — so its elapsed time effectively represents the **total GPU execution time**.

The execution steps included within `Trap_wrapper` are:

- Initialization of `*trap_p` with the starting sum (Line 29).
- Calculation of the step size $h = (b - a)/n$ (Line 30).
- Launching of the CUDA kernel `Dev_trap`, where each thread computes its partial contribution.
- The final update of `*trap_p` after all threads finish (Line 35).

Hence, by placing timing calls immediately **before and after** the call to `Trap_wrapper`, we capture the **total elapsed time** of the GPU-based trapezoidal computation.

The timing is measured using the `GET_TIME` macro (defined in `timer.h`), which records wall-clock time:

As noted in earlier performance discussions (Section 2.6.4), multiple timing trials should be conducted, and the **minimum** time is typically reported. However, if only a few results fall within 1% of the minimum, that value may not be reproducible. In such cases, it's better to report the **mean or median** elapsed time to provide a more representative and stable measure of performance.

In the experiments described, both the serial and CUDA trapezoidal rule implementations were executed repeatedly, and **mean times** (averaged over at least 50 runs) were reported.

For example, using:

- $n = 2^{20} = 1,048,576$ trapezoids
- $f(x) = x^2 + 1$
- Interval $[a, b] = [-3, 3]$
- $1024 \text{ blocks} \times 1024 \text{ threads per block} = 1,048,576 \text{ total threads}$

The results in **Table 6.5** show:

- The **GK20A GPU** (192 SPs) was significantly faster than an **ARM Cortex-A15 CPU**.
- However, a single **Intel Core i7** core was still faster than the GK20A GPU.
- The **Titan X GPU** (3072 SPs) outperformed the Intel core by **about 45%**, though the gain was less than expected given the large number of streaming processors (SPs).

These results highlight that **raw hardware parallelism** alone does not guarantee proportional speedup — **memory bandwidth, kernel launch overhead, and arithmetic intensity** all influence the effective performance of CUDA programs like the trapezoidal rule.

5.11 CUDA trapezoidal rule II: improving performance

5.11.1 Tree-structured communication

We can visualize the execution of the *global sum* in the CUDA trapezoidal rule as a more or less random, linear ordering of the threads. For example, consider 8 threads within a single block numbered **0–7**. One of the threads will be the first to succeed with the call to `atomicAdd`; suppose it's thread 5. Then another thread—say thread 2—succeeds next, and so on. Continuing in this way, we get a sequence of atomic additions, one per thread. **Table 6.6** illustrates how this sequence might progress over time.

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

In this example, to keep the computation simple, assume $f(x) = 2x + 1$, $a = 0$, and $b = 8$. Hence,

$$h = \frac{8 - 0}{8} = 1,$$

and the initial value of `trap_p` is:

$$0.5 \times (f(a) + f(b)) = 0.5 \times (1 + 17) = 9.$$

What's important to note is that this approach may effectively **serialize** the threads since each thread must wait for its turn to perform its `atomicAdd`. The computation then requires a sequence of 8 consecutive additions. **Figure 6.3** illustrates one possible order of execution.

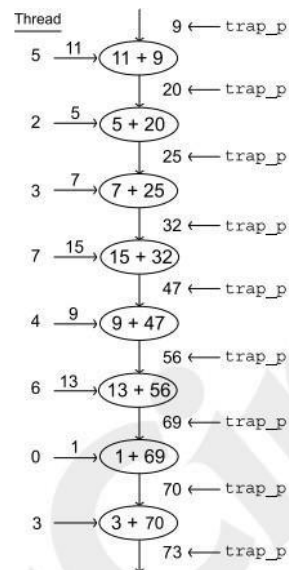


FIGURE 6.3
Basic sum.

To improve efficiency, instead of allowing each thread to wait for its turn to add its result into `*trap_p`, threads can be **paired** so that half of the active threads add their partial sum to their partner's sum. This yields a **tree-structured pattern** (or "shrub-like" structure) as shown in Figure 6.4.

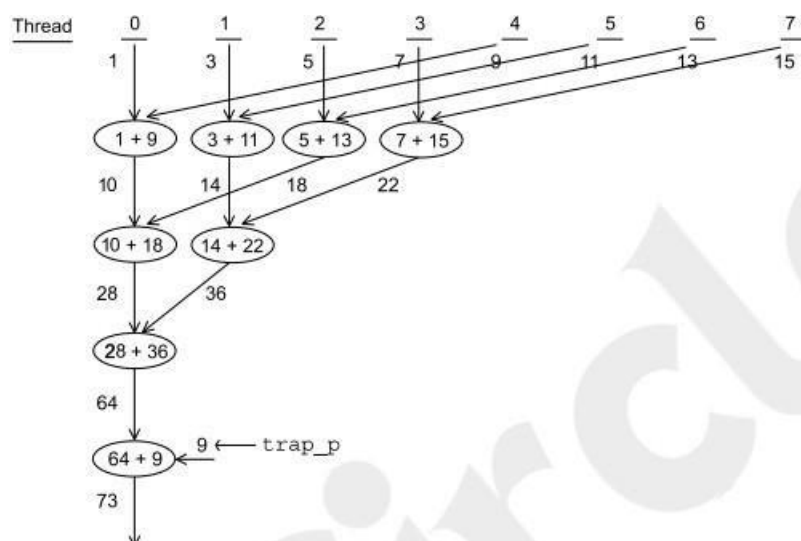


FIGURE 6.4
Tree-structured sum.

In the figures, we've gone from needing 8 consecutive additions (linear sequence) to only 4 (tree sequence). More generally:

- With the basic approach, if we double the number of threads (e.g., $8 \rightarrow 16$), we double the number of additions required.
- With the tree-structured approach, we add only one more step to the sequence.

Thus, with `tthreads` and `tvalues`:

- The **basic approach** requires t additions.
- The **tree-structured approach** requires $\log_2(t) + 1$ additions.

For example:

- $t = 1000$: from 1000 to 11 additions.
- $t = 1,000,000$: from 1,000,000 to 21 additions.

There are two standard CUDA implementations of the tree-structured sum:

1. **Shared Memory Implementation:**

Used on devices with compute capability < 3 , where threads within a block store and reduce partial sums in shared memory.

2. **Warp Shuffle Implementation:**

For devices with compute capability ≥ 3 , CUDA provides **warp shuffle** functions that allow threads within a warp to directly access other threads' data. This approach eliminates the need for shared memory and synchronization barriers, providing faster reductions.

5.11.2 Local variables, registers, shared and global memory

NVIDIA GPU has access to two main types of memory:

- **Shared memory**, which is private to the SM and accessible only by the threads within that SM. More precisely, the shared memory allocated for a *thread block* can be accessed only by threads belonging to that block.
- **Global memory**, which is accessible to all SMs and all threads in the grid.

The key distinction lies in **speed** and **capacity**. Shared memory is relatively small but very fast, while global memory is large but considerably slower. Thus, GPU memory can be viewed as a **three-level hierarchy**:

1. **Registers** – the fastest and smallest level.
2. **Shared memory** – intermediate in both speed and size.
3. **Global memory** – the largest but slowest level.

Table 6.7 provides comparative data illustrating these relative sizes. Typically, accessing data stored in registers takes about **1 clock cycle**. Moving data between shared memory locations can take roughly an **order of magnitude** longer, while global memory accesses may take **two to three orders of magnitude** more time.

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

This dramatic difference explains why optimizing memory usage is crucial for CUDA performance.

A natural question arises: *what about local variables?* Where are they stored, and how quickly can they be accessed?

The answer depends on the availability of registers and total memory usage. When sufficient registers are available, local variables are stored directly in **registers**, giving maximum speed. However, if register space is exhausted, the compiler automatically **spills** local variables to a special portion of **global memory** reserved for each thread — often called **local memory**. Although thread-private, this spilled memory inherits the **latency and access cost of global memory**.

Therefore, as long as there is enough register storage, a kernel's performance generally improves when it maximizes **register usage** and minimizes reliance on **shared** and especially **global** memory. The trade-off, however, is that register capacity is very limited compared to the much larger pools of shared and global memory.

5.11.3 Warps and warp shuffles

If we can carry out the **global sum entirely within registers**, the performance will naturally be superior to using shared or global memory. This is made possible through the **warp shuffle functions** introduced in CUDA devices with **compute capability 3.0** and higher.

In CUDA, a **warp** is a group of **threads with consecutive ranks** within a thread block. The number of threads in a warp is currently **32**, although NVIDIA has indicated this value may change in future architectures. The system provides a predefined variable:

```
int warpSize;
```

Threads in a warp execute in **SIMD (Single Instruction, Multiple Data)** fashion. This means that threads across different warps can execute different instructions simultaneously without penalty, but all threads within a single warp must execute the same instruction. When threads within a warp take **different control paths**, such as opposite branches of an if-else statement, they **diverge**. Once they complete their respective paths and resume executing the same instruction, they **converge**.

The **rank** of a thread within its warp is known as its **lane**, computed as:

```
lane = threadIdx.x % warpSize;
```

The **warp shuffle functions** enable a thread to directly read values from the **registers of another thread** in the same warp. This avoids slower shared or global memory accesses. For the tree-structured sum, the relevant function is:

```
__device__ float __shfl_down_sync(
    unsigned mask,    /* input */
    float var,        /* input */
    unsigned diff,    /* input */
    int width = warpSize /* input */
);
```

The **mask** specifies which threads are participating in the shuffle. Each bit corresponds to a thread's **lane ID**. For safe execution, all participating threads must have **converged**—that is, reached the call together—before any executes the shuffle. In most cases, **all threads** in the warp participate, so we define:

```
unsigned mask = 0xffffffff;
```

Here, 0x indicates a hexadecimal literal, and 0xffffffff corresponds to **32 binary 1s**, meaning **all 32 threads** are active in the operation.

When a thread with **lane l** calls `__shfl_down_sync`, it receives the value of `var` from the thread with:

```
lane = l + diff;
```

Since `diff` is an unsigned integer (non-negative), the operation retrieves data from a **higher-ranked thread**, hence the name “**shuffle down**.”

The width parameter specifies the number of threads participating in the operation, but as it defaults to `warpSize`, it is usually omitted.

There are, however, some **potential pitfalls**:

- If thread `l` calls `__shfl_down_sync` but **thread `l + diff`** does not, the returned value is **undefined**.
- If $l + \text{diff} \geq \text{warpSize}$, the call returns the value already stored in `var` on thread `l`.
- If the **thread block size is not a multiple of `warpSize`**, the last warp will have fewer than 32 threads. Suppose the last warp has `w` threads ($0 < w < \text{warpSize}$); then if $l + \text{diff} \geq w$, the result is also undefined.

To ensure well-defined and efficient behavior, it is best to:

- Have **all threads** in a warp call `__shfl_down_sync`.
- Choose a **block size** that is a **multiple of `warpSize` (32)**.

This guarantees that every warp participates fully in the operation and prevents undefined results during register-level communication.

5.11.4 Implementing tree-structured global sum with a warp shuffle

A **tree-structured global sum** can be efficiently implemented within a warp using registers and the warp shuffle function. The implementation in CUDA is shown below:

```
__device__ float Warp_sum(float var) {
    unsigned mask = 0xffffffff;
    for (int diff = warpSize / 2; diff > 0; diff = diff / 2)
        var += __shfl_down_sync(mask, var, diff);
    return var;
} /* Warp_sum */
```

This function progressively reduces the number of active threads performing additions by half in each iteration, forming a **binary tree structure** of sums. As illustrated in **Fig. 6.5**, the process can be visualized using a smaller warp size of 8 for clarity.

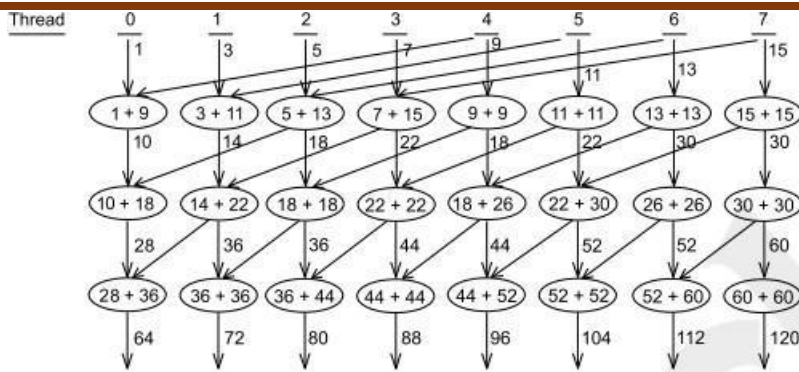


FIGURE 6.5

Tree-structured sum using warp shuffle.

At each step, the `__shfl_down_sync` function shifts the values of `var` from threads with higher lane IDs to those with lower ones, and the lower-lane threads accumulate these values. For instance, when `diff = 4`, threads with **lane IDs 0–3** receive values from threads **4–7**. However, for threads with **lane IDs 4–7**, since $1 + 4 \geq 8$, the function simply returns their own `var` value. Hence, their values get doubled at that stage. This behavior repeats in later iterations—for example, at `diff = 2` for lanes 6 and 7, and at `diff = 1` for lane 7—where only one arrow enters the summing node in Fig. 6.5.

From a **practical standpoint**, this implementation produces the **correct total sum only on the thread with lane ID 0**. The remaining threads will hold partial or redundant sums. If the final result needs to be available to **all threads** within the warp, an alternative function such as `__shfl_xor` can be used.

6.11.5 Shared memory and an alternative to the warp shuffle

If the GPU's compute capability is below 3.0, warp shuffle functions cannot be used since threads cannot access one another's registers directly. Instead, threads in the same block can communicate through **shared memory**, which, though slower than register access, can yield similar performance when properly managed.

Because threads in a warp execute synchronously, shared memory can mimic the behavior of warp shuffle operations. The following function demonstrates this:

```
__device__ float Shared_mem_sum(float shared_vals[]) {
    int my_lane = threadIdx.x % warpSize;
    for (int diff = warpSize / 2; diff > 0; diff /= 2) {
        int source = (my_lane + diff) % warpSize; // Ensure 0 <= source < warpSize
        shared_vals[my_lane] += shared_vals[source];
    }
    return shared_vals[my_lane];
}
```

All threads within a warp should invoke this function, with `shared_vals` located in the shared memory of the SM executing the warp. Since threads operate in **SIMD fashion**, updates occur in

lockstep, eliminating race conditions—each thread reads from `shared_vals[source]` before writing to `shared_vals[my_lane]`.

This approach is known as a **dissemination sum** (or dissemination reduction) rather than a traditional tree-structured sum. **Figure 6.6** illustrates the series of copy and addition operations that occur, though it omits the direct contributions of each thread for clarity. Every thread retrieves a value from another thread in each iteration, and by the end, all threads possess the same final sum—not just thread 0.

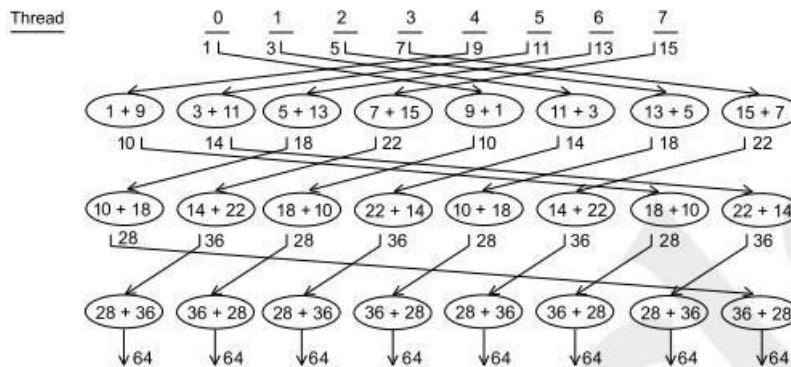


FIGURE 6.6

Dissemination sum using shared memory.

Although this full-warp summation is unnecessary for the trapezoidal rule, it can be advantageous in other algorithms. When a warp is active, all its threads either execute the same instruction or remain idle, so uniform instruction execution does not impose additional cost.

The `Shared_mem_sum` function itself does not require shared memory; its argument `shared_vals` could reside in either **global** or **shared memory**. However, performance improves when it is defined as a `__shared__` variable within the kernel, for instance:

```
__shared__ float shared_vals[32];
```

This reserves 32 float locations in the shared memory of each SM for its corresponding thread block.

If the shared memory size is not known at compile time, it can be declared dynamically:

```
extern __shared__ float shared_vals[];
```

When launching the kernel, the third parameter in the triple-angle brackets specifies the required shared memory size in bytes. For example, in a trapezoidal rule kernel `Dev_trap`, it can be invoked as:

```
Dev_trap<<<blk_ct, th_per_blk, th_per_blk * sizeof(float)>>>(...args...);
```

This allocates `th_per_blk` floats in the shared memory for each thread block, ensuring efficient summation across threads.

5.12 Implementation of trapezoidal rule with warpSize thread blocks

After exploring the use of **efficient summation techniques**, **warps**, **warp shuffles**, and **shared memory**, we can now develop improved CUDA implementations of the **trapezoidal rule**. In both implementations, each **thread block** consists of warpSize threads, and a **tree-structured sum** is employed to combine the results within each warp. Once the threads in a warp compute their respective function values and accumulate their local results, the **thread with lane ID 0** adds the warp's partial sum to the global total using the atomicAdd operation.

5.12.1 Host Code

For both the **warp shuffle** and **shared memory** versions, the **host code** remains nearly identical to that of the initial CUDA implementation. The main distinction is that there is **no th_per_blk variable**, as each thread block is now assumed to contain exactly warpSize threads. This simplifies the host configuration and ensures that each block aligns perfectly with the hardware's warp execution model, allowing the improved summation strategies to operate efficiently.

5.12.2 Kernel with warp shuffle

In **Program 6.13**, the kernel improves upon the earlier trapezoidal rule implementation by incorporating **warp-level summation** instead of having each thread update the global sum individually. The variable my_trap is initialized just as in the previous version (Program 6.12), representing the partial contribution computed by each thread.

```

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      float my_trap = 0.0f;
10     if (0 < my_i && my_i < n) {
11         float my_x = a + my_i*h;
12         my_trap = f(my_x);
13     }
14
15     float result = Warp_sum(my_trap);
16
17     /* result is correct only on thread 0 */
18     if (threadIdx.x == 0) atomicAdd(trap_p, result);
19 } /* Dev_trap */

```

Program 6.13: CUDA kernel implementing trapezoidal rule and using Warp_sum.

However, rather than directly adding each thread's contribution to the global sum (*trap_p), all threads within a **warp** (or equivalently here, a **thread block**) first call the **Warp_sum** function. This function performs a **tree-structured reduction** across the threads in the warp, producing a single sum value — the **warp sum**.

After this reduction step, only the **thread with lane ID 0** (the first thread in the warp) updates the global accumulator by adding the warp sum to *trap_p. Because multiple thread blocks can execute concurrently and attempt to update *trap_p at the same time, a **race condition** would arise if these updates were not synchronized.

To prevent this, the kernel employs **atomicAdd**, ensuring that each warp's contribution is safely and correctly accumulated into the global total without interference from other warps or thread blocks. This atomic operation guarantees data consistency while allowing all warps to execute in parallel.

5.12.3 Kernel with shared memory

In **Program 6.14**, the kernel uses **shared memory** instead of warp shuffle for summing thread results within a warp. It declares a shared array using `__shared__ float shared_vals[WARPSZ];` where `WARPSZ` is defined as `#define WARPSZ 32` since `warpSize` isn't known at compile time.

```

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 shared_vals[WARPSZ];
8      int my_i = blockDim.x * blockIdx.x + threadIdx.x;
9      int my_lane = threadIdx.x % warpSize;
10
11     shared_vals[my_lane] = 0.0f;
12     if (0 < my_i && my_i < n) {
13         float my_x = a + my_i*h;
14         shared_vals[my_lane] = f(my_x);
15     }
16
17     float result = Shared_mem_sum(shared_vals);
18
19     /* result is the same on all threads in a block. */
20     if (threadIdx.x == 0) atomicAdd(trap_p, result);
21 } /* Dev_trap */

```

Program 6.14: CUDA kernel implementing trapezoidal rule and using shared memory.

Each thread stores its partial sum in `shared_vals`, and the **Shared_mem_sum** function performs the reduction. Finally, **lane 0** adds the warp's total to the global sum using `atomicAdd` to avoid race conditions. This version functions like the warp shuffle implementation but uses shared memory for intra-warp communication.

5.12.4 Performance

To evaluate performance (Table 6.8), the same integration problem as before (Table 6.5) is used: $f(x) = x^2 + 1$ over the interval $[-3, 3]$ with $2^{20} = 1,048,576$ trapezoids.

Since each thread block has 32 threads, the setup uses **32,768 thread blocks** ($32 \times 32,768 = 1,048,576$). This configuration allows comparison of different implementations—original, warp shuffle, and shared memory—for performance efficiency.

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
SMTs, 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

5.13 CUDA trapezoidal rule III: blocks with more than one warp

In CUDA programming, a **race condition** occurs when multiple threads try to access or modify the same data simultaneously, and the final result depends on the order of execution.

In the case discussed, the race condition arises when **warp 0** tries to compute the total of the warp sums in a block **before** all other warps have finished their individual sums. For example, if warp 0 finishes early and starts reading warp 1's partial sum before warp 1 is done, the result will be **incorrect**.

5.13.1 `__syncthreads`

CUDA solves this problem using a **barrier synchronization function** called `__syncthreads()`.

This function ensures that **all threads in a block** reach the same point before any of them proceed further. When `__syncthreads()` is used, warp 0 will wait until all other warps have completed their work, thus eliminating the race condition.

The correct pseudocode for computing the block sum is:

```
Each thread computes its contribution;
Each warp adds its threads' contributions;
__syncthreads();
Warp 0 in block adds warp sums;
```

After inserting `__syncthreads()`, the addition of warp sums begins **only after** every warp has finished computing its partial sum.

Two important points to remember:

1. **All threads in a block must execute `__syncthreads()`** — if only a subset of threads reach it (for example, due to an if condition), those threads will wait forever, causing the kernel to hang.
2. **`__syncthreads()` works only within a block**, not across multiple blocks. Threads in different blocks execute independently and cannot synchronize using this function.

Hence, `__syncthreads()` is essential for correct coordination of threads within a block, especially when sharing intermediate results through shared memory.

5.13.2 More shared memory

When implementing the pseudocode in actual CUDA code, an additional detail must be handled: **how warp 0 accesses the results computed by other warps after the `__syncthreads()` call.**

Warp shuffle instructions allow threads to share register values, but **only within the same warp**. Threads in **different warps cannot directly access each other's registers**. Hence, the final block-level reduction cannot rely on warp shuffle alone.

The solution is to use **shared memory**, which can be accessed by all threads in a block. Each warp can store its partial sum into a shared memory array. After synchronization, warp 0 can read these values to compute the total block sum.

```
__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
```

5.13.3 Shared memory warp sums

When using shared memory instead of warp shuffles, we need enough space for all threads in a block. So we declare:

```
cuda

#define MAX_BLKSZ 1024
__shared__ float thread_calcs[MAX_BLKSZ];
```

Each warp uses a subarray of `thread_calcs` for its calculations:

```
c

float *shared_vals = thread_calcs + my_warp * warpSize;
shared_vals[my_lane] = f(my_x);
float my_result = Shared_mem_sum(shared_vals);
```

A race condition occurs if all warps try to store their warp sums in the same shared region (e.g., `thread_calcs[w]`). To avoid this, synchronization is needed:

```
__syncthreads();
if (my_lane == 0)
    thread_calcs[my_warp] = my_result;
__syncthreads();
if (my_warp == 0)
    my_result = Shared_mem_sum(thread_calcs);
```

However, frequent `__syncthreads()` calls reduce performance. A better solution is for each warp to store its sum in the **first element of its own subarray**:

```
if (my_lane == 0)
    shared_vals[0] = my_result;
__syncthreads();
```

This is safe because each warp writes to a different part of shared memory, so no overlap or race condition occurs.

5.13.4 Shared Memory Banks

Shared memory in each Streaming Multiprocessor (SM) is divided into **32 banks** (16 for older GPUs). Each bank can be accessed by one thread per warp simultaneously. If two or more threads access **different locations within the same bank**, the accesses are **serialized**, reducing performance.

Table 6.9 illustrates this structure: consecutive elements of `thread_calcs` are distributed across the 32 banks. When threads access elements like 0, 32, 64, ..., 992, all these map to the **same bank**, causing serialization and slower performance.

- If each thread in a warp accesses a **different bank**, access is parallel.
- If multiple threads access **different locations** in the same bank, access is serialized.
- If multiple threads read the **same location**, the read is broadcast efficiently.

To optimize access, warp sums should be stored in **contiguous memory** (e.g., a row of the bank layout).

Hence, we declare two shared arrays to separate thread and warp sums:

```
__shared__ float thread_calcs[MAX_BKLSZ];
__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();
```

5.13.5 Finishing Up

After all warps compute their sums, **warp 0** adds up the values in `warp_sum_arr`. Finally, **thread 0** adds the block's total to the global sum using `atomicAdd`:

```
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);
```

This ensures correctness even when the number of warps in a block is less than 32.

5.13.6 Performance

The shared memory and warp shuffle versions both improve performance by reducing the number of calls to `atomicAdd`.

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 SMs, SPs	2.3 GHz	852 MHz 1, 192	3.5 GHz	1.08 GHz 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