UNIT-1
INTRODUCTION

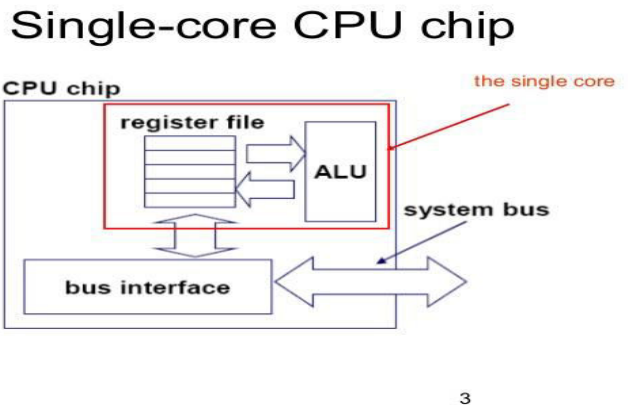
What is Multi Core Architecture?

When a processor has more than one core to execute all the necessary functions of a computer, it's processor is known to be a multi core architecture. In other words, a chip with more than one CPU(Central Processing Unit).

What is the difference between a single core processor and a multi core processor?

SINGLE-CORE PROCESSORS

Single core processors have only one processor in die to process the instructions. A **single core** is a **single** calculation unit or processing unit that executes calculations. Dual **core** means a cpu with two calculation units or two processing units. The difference in performance of dual **core** and **single core** varies on the software and how much software you are running on your compute



3

Fig. 1 Single core Processor Architecture

Problems of Single Core Processors

- As we try to increase the clock speed of this processor, the amount of heat produced by the chip also increases. It is a big hindrance in the way of single core processors to continue evolving.

MULTI-CORE PROCESSORS

Multicore processors are the latest processors which became available in the market after 2005. These processors use two or more cores to process instructions at the same time by using hyper threading. The multiple cores are embedded in the same die. The multicore processor may look like a single processor but it contains two (dual-core), three (tricore), four (quad-core), six (hexa-core), eight (octa-core) or ten (deca-core) cores. Some processors even have 22 or 32 cores.

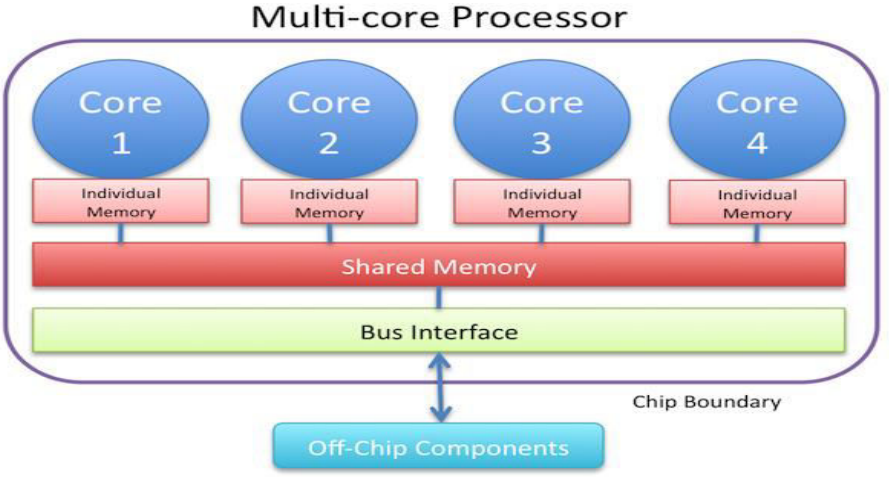


Fig.2 Multicore Processor Architecture

Advantages of multi core CPU

- The largest boost in performance will likely be noticed in improved response-time while running CPU intensive processes, like anti-virus scans, ripping/burning media.
- Aassuming that the die can fit into the package, physically, the multi-core CPU designs require much less printed Circuit Board(PCB) space than multi-chip SMP designs.
- Also, a dual core processor uses slightly less power than two coupled single core processors, principally because of the decreased power required to drive signals external to the chip

Problems with multicore processors

- According to Amdahl's law, the performance of parallel computing is limited by its serial components. So, increasing the number of cores may not be the best solution.There is need to increase the clock speed of individual cores.

Type of CPU	Description	Capabilities
Single Core CPU	Has one core to process different operations; microprocessors were single cores from the early 1970s on	Word processing, checking email, surfing the Internet, watching videos
Dual Core CPU	Has two cores to process operations; able to process more information at the same time	Flash-enabled web browsing, video and conference chatting
Quad Core CPU	Contains two dual core processors in one integrated circuit	Voice-GPS systems, multi-player gaming, video editing

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.

- Single instruction, multiple data, or SIMD, systems are parallel systems.
- 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 ith ALU, have the ith 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 synchronously,

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 simple loops that operate on large arrays of data.

- Parallelism that's obtained by dividing data among the processors and having the processors all apply (more or less) the same instructions to their subsets of the data is called data-parallelism.
- SIMD parallelism can be very efficient on large data parallel problems, but SIMD systems often don't do very well on other types of parallel problems.
- SIMD systems have had a somewhat checkered history. In the early 1990s amaker of SIMD systems (Thinking Machines) was the largest manufacturer of parallel supercomputers.
- However, by the late 1990s the only widely produced SIMD systems were **vector processors**.
- More recently, graphics processing units, or GPUs, and desktop CPUs are making use of aspects of SIMD computing.

Vector processors

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

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

Note: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 greatvirtue 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(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.

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.

MIMD systems

- ✓ Multiple instruction, multiple data, or MIMD, systems support multiple simultaneousinstruction 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.
- ✓ There are two principal types of MIMD systems: **shared-memory systems** and **distributed-memory systems**.

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.

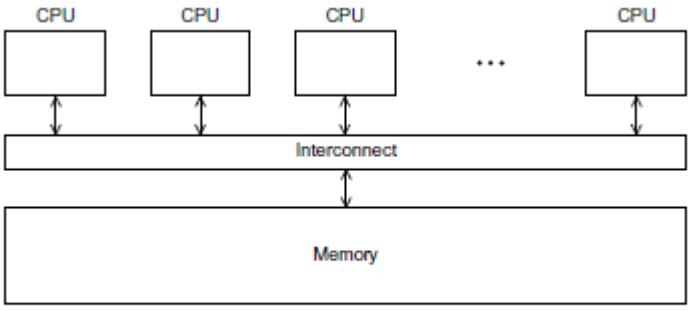


FIGURE 2.3
A shared-memory system

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.

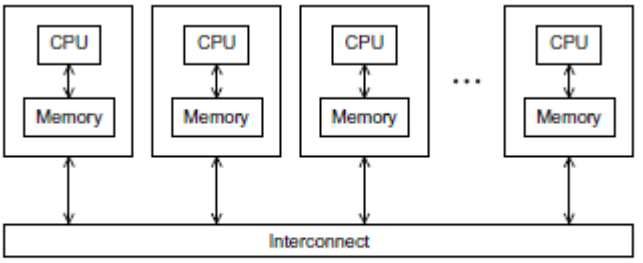


FIGURE 2.4
A distributed-memory system

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. 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**.

Buses

- ✓ 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.

Switched Interconnects

- ✓ Switched interconnects use switches to control the routing of data among the connected devices.

Crossbar

- ✓ The lines are bidirectional communication links, the squares are cores or memory modules, and the circles are switches.
- ✓ The individual switches can assume one of the two configurations shown in 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. 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 P1 writes to M4, P2 reads from M3, P3 reads from M1, and P4 writes to M2.

- ✓ 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.

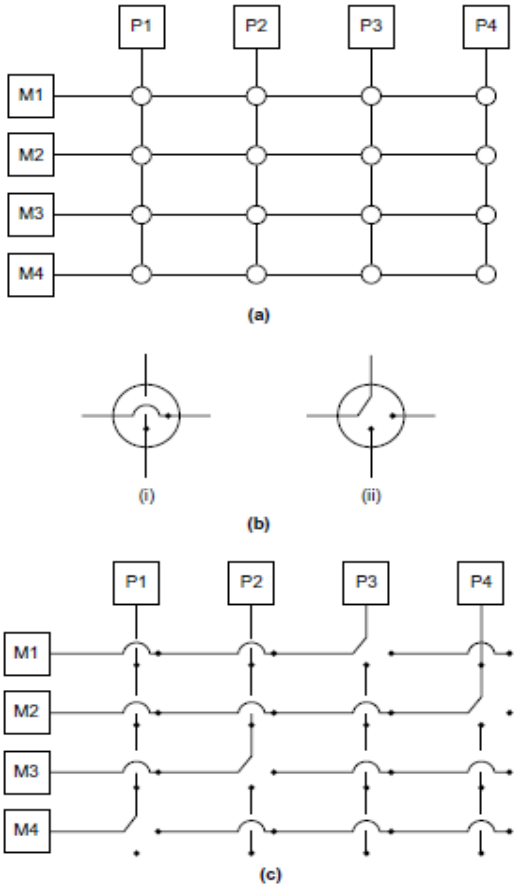


FIGURE 2.7
(a) A crossbar switch connecting four processors (P_i) and four memory modules (M_j); (b) configuration of internal switches in a crossbar; (c) simultaneous memory accesses by the processors

Distributed-memory interconnects

Distributed-memory interconnects are often divided into two groups:**direct interconnects** and **indirect interconnects**.

Direct interconnect

- ✓ 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 divided into two halves, and each half contains half of the processors or nodes. 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 = q2 nodes (where q is even), then we can split the nodes into two halves by removing the "middle" horizontal links and the "wrap around" horizontal links. See Figure 2.10. This suggests that the bisection width is at most $2q = \sqrt{p}$. In fact, this is the smallest possible number of links and the bisection width of a square two-dimensional toroidal mesh is $\sqrt{2p}$.
- ✓ 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.

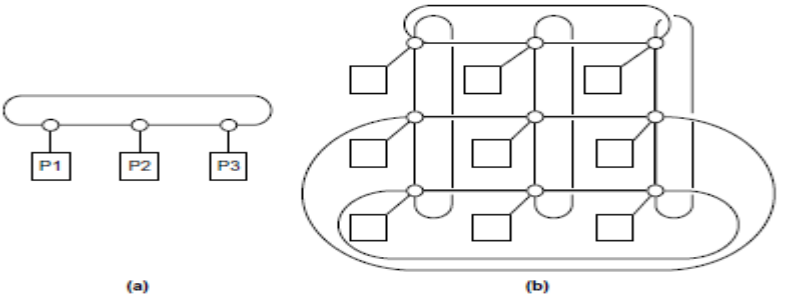


FIGURE 2.8
(a) A ring and (b) a toroidal mesh

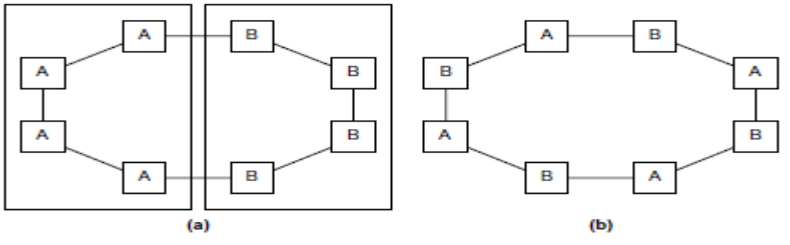


FIGURE 2.9
Two bisections of a ring: (a) only two communications can take place between the halves and (b) four simultaneous connections can take place

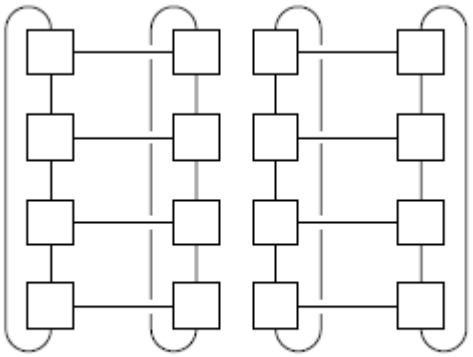


FIGURE 2.10
A bisection of a toroidal mesh

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.

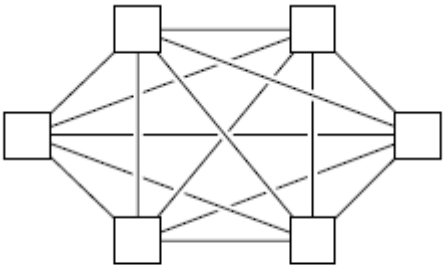


FIGURE 2.11
A fully connected network

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.

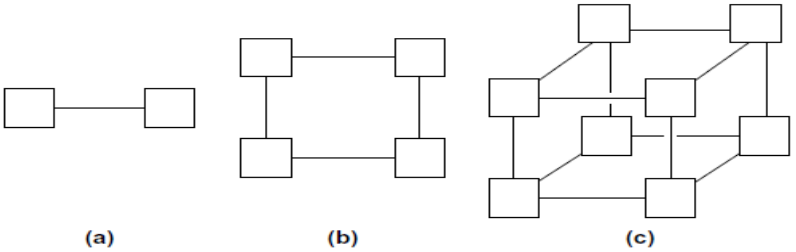


FIGURE 2.12
(a) One-, (b) two-, and (c) three-dimensional hypercubes

Indirect interconnects

- ✓ They 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.

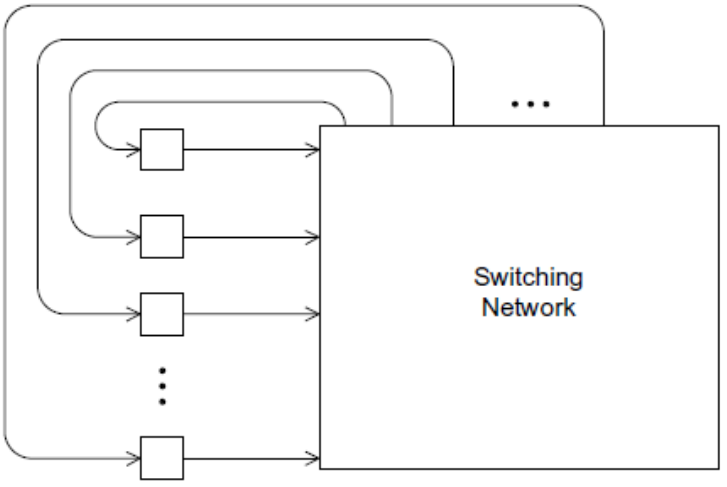


FIGURE 2.13

A generic indirect network

- ✓ The crossbar and the omega network are relatively simple examples of indirect networks.

- ✓ 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.

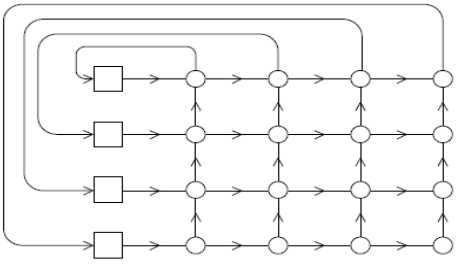


FIGURE 2.14

A crossbar interconnect for distributed-memory

- ✓ An omega network is shown in Figure 2.15. The switches are two-by-two crossbars (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 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 ½ 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.

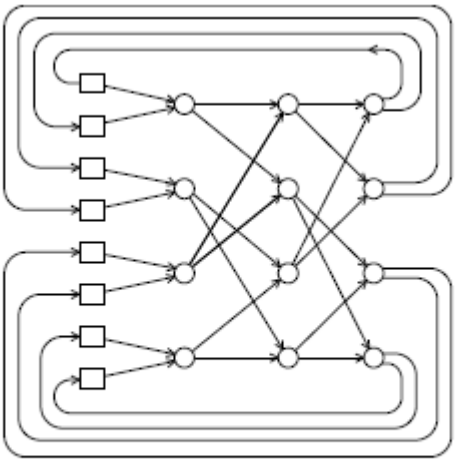


FIGURE 2.15
An omega network

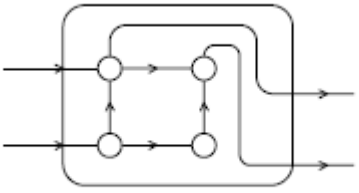


FIGURE 2.16
A switch in an omega network

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

Message transmission time= l+n/b

DISTRIBUTED AND SHARED MEMORY ARCHITECTURES

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 ashared-memory system, the processors usually communicate implicitly by accessing shared data structures.

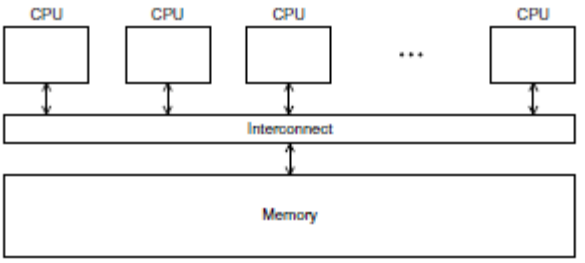


FIGURE 2.3
A shared-memory system

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.

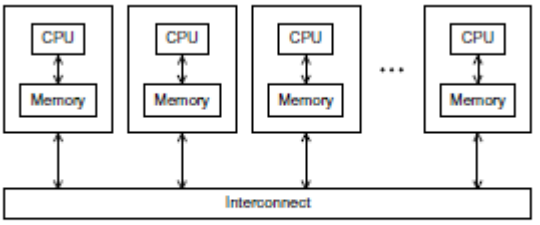


FIGURE 2.4
A distributed-memory system

Shared-memory systems

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

- ✓ In shared-memory systems with multiple multicore processors, the interconnect can either connect all the processors directly to main memory or each processor can have a direct connection to a block of main memory, and the processors can access each 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.

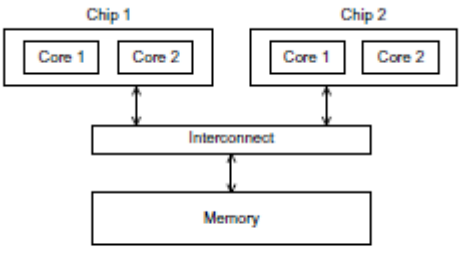


FIGURE 2.5
A UMA multicore system

- ✓ 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.

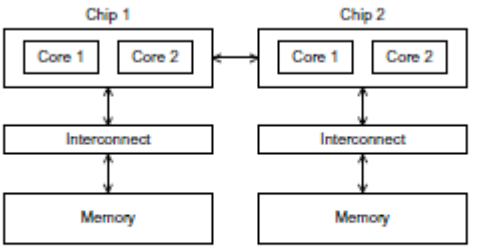


FIGURE 2.6
A NUMA multicore system

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 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.

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, y0 is private and owned by core 0, and y1 and z1 are private and owned by core 1. Now suppose the following statements are executed at the indicated times:

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

- ✓ Then the memory location for y0 will eventually get the value 2, and the memory location for y1 will eventually get the value 6.

- ✓ However, it's not so clear what value z1 will get. It might at first appear that since core 0 updates x to 7 before the assignment to z1, z1 will get the value 4*7= 28.
- ✓ However, at time 0, x is in the cache of core 1. So unless for some reason x is evicted from core 0's cache and then reloaded into core 1's cache, it actually appears that the original value x = 2 may be used, and z1 will get the value 4*2= 8.

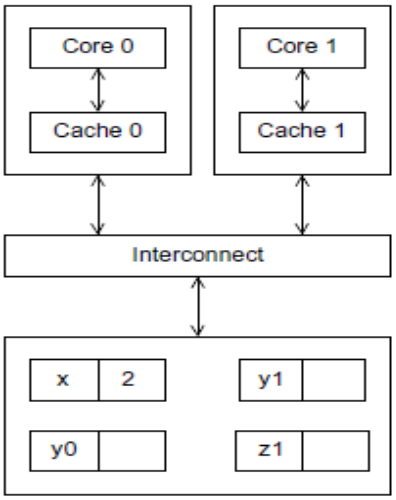


FIGURE 2.17

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.
- ✓ When **shared data** are cached, the shared value may be replicated in multiple caches. In addition to the reduction in access latency and required memory bandwidth, this replication also provides a reduction in contention that may exist for shared data items that are being read by multiple processors simultaneously. ***This problem is called as cache coherence.***

There are two main approaches to insuring cache coherence:

- ➢ snooping cache coherence
- ➢ directory-based cache coherence.

Snooping cache coherence

The idea behind snooping comes from bus-based systems:

- ❖ When the cores share a bus, any signal transmitted on the bus can be "seen" by all the cores connected to the bus.
- ❖ Thus, when core 0 updates the copy of x stored in its cache, if it also broadcasts this information across the bus, and if core 1 is "snooping" the bus, it will see that x has been updated and it can mark its copy of x as invalid. This is more or less how snooping cache coherence works.

- The principal difference between our description and the actual snooping protocol is that the broadcast only informs the other cores that the cache line containing x has been updated, not that x has been updated.

A couple of points should be made regarding snooping.

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

Directory-based cache coherence

- Unfortunately, in large networks broadcasts are expensive, and snooping cache coherence requires a broadcast every time a variable is updated .
- So snooping cache coherence isn't scalable, because for larger systems it will cause
  performance to degrade.
- For example, suppose we have a system with the basic distributed-memory architecture (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.
- 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.

PERFORMANCE

➢ 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 $T_{parallel} = T_{serial}/p$. When this happens, we say

that our parallel program has **linear speedup**.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..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

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

- ✓ 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*parallel = *T*serial/*p* + *T*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.
- ✓ A final issue to consider is what values of *T*serial should be used when reporting speedups and efficiencies.

➢ **Amdahl's law**

Gene Amdahl made an observation 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 serialprogram. 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 Tserial= 20 seconds, then the run-time of the parallelized part will be 0.9*Tserial/p = 18/p and
- ✓ the run-time of the "unparallelized" part will be 0.1*Tserial =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*Tserial/p =18/p gets closer and closer to 0, so the total parallel run-time can't be smaller than 0.1_Tserial D 2. That

is, the denominator in S can't be smaller than 0.1_Tserial D 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.

➢ **Scalability**

- ✓ The word **"scalable"** has a wide variety of informal uses. 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 Tserial =n, where the units of Tserial are in microseconds, and n is also the problem size. Also suppose that Tparallel = 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

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

➢ **Taking timings**

There are a *lot* ofdifferent 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 usuallydifferent

❖ 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 takesto 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.

PARALLEL PROGRAM DESIGN

Ian Foster provides an outline of steps in his online book Designing and Building Parallel Programs

1. **Partitioning.** Divide the computation to be performed and the data operated on by the computation into small tasks. The focus here should be on identifying tasks that can be executed in parallel.

2. **Communication**. Determine what communication needs to be carried out among the tasks identified in the previous step.
3. **Agglomeration or aggregation**. Combine tasks and communications identified in the first step into larger tasks. For example, if task A must be executed before task B can be executed, it may make sense to aggregate them into a single composite task.
4. **Mapping**. Assign the composite tasks identified in the previous step to processes/threads. This should be done so that communication is minimized, and each process/thread gets roughly the same amount of work.This is sometimes called Foster's methodology.

An example

Suppose we have a program that generates large quantities of floating point data that it stores in an array. In order to get some feel for the distribution of the data, we can make a histogram of the data. Recall that to make a histogram, we simply divide the range of the data up into equal sized subintervals,or *bins*, determine the number of measurements in each bin, and plot a bar graph showing the relative sizes of the bins. As a very small example, suppose our data are
1.3, 2.9, 0.4, 0.3, 1.3, 4.4, 1.7, 0.4, 3.2, 0.3, 4.9, 2.4, 3.1, 4.4, 3.9, 0.4, 4.2, 4.5, 4.9, 0.9.

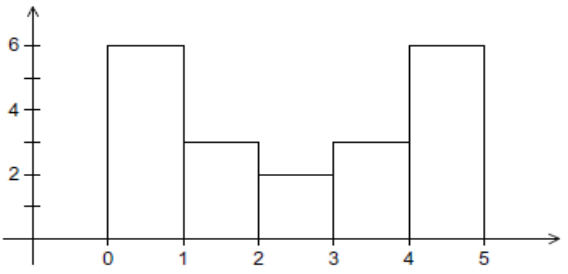


FIGURE 2.20
A histogram

A serial program

It's pretty straightforward to write a serial program that generates a histogram. We need to decide what the bins are, determine the number of measurements in each bin, and print the bars of the histogram. Since we're not focusing on I/O, we'll limit ourselves to just the first two steps, so the input will be
1. the number of measurements, data count;
2. an array of data count floats, data;
3. the minimum value for the bin containing the smallest values, min meas;
4. the maximum value for the bin containing the largest values, max meas;
5. the number of bins, bin count;
The output will be an array containing the number of elements of data that lie in each bin. To make things precise, we'll make use of the following data structures:

- `bin_maxes.` An array of `bin_count` floats
- `bin_counts.` An array of `bin_count` ints

The array bin maxes will store the upper bound for each bin, and bin counts will store the number of data elements in each bin. To be explicit, we can define

```
bin_width = (max_meas – min_meas)/bin_count
```

Then bin maxes will be initialized by

```
for (b = 0; b < bin_count; b++)
   bin_maxes[b] = min_meas + bin_width*(b+1);
```

We'll adopt the convention that bin b will be all the measurements in the range

```
bin_maxes[b-1] <= measurement < bin_maxes[b]
```

Of course, this doesn't make sense if $b = 0$, and in this case we'll use the rule that bin 0 will be the measurements in the range

```
min_meas <= measurement < bin_maxes[0]
```

This means we always need to treat bin 0 as a special case, but this isn't too onerous. Once we've initialized bin maxes and assigned 0 to all the elements of bin_counts, we can get the counts by using the following pseudo-code:

```
for (i = 0; i < data_count; i++) {
   bin = Find_bin(data[i], bin_maxes, bin_count, min_meas);
   bin_counts[bin]++;
}
```

The Find_bin function returns the bin that data[i] belongs to. This could be a simple linear search function: search through bin maxes until you find a bin b that satisfies

```
bin_maxes[b-1] <= data[i] < bin_maxes[b]
```

Parallelizing the serial program

If we assume that data count is much larger than bin count, then even if we use binary search in the Find bin function, the vast majority of the work in this code will be in the loop that determines the values in bin counts. The focus of our parallelization should therefore be on this loop, and we'll apply Foster's methodology to it.

- For the first step we might identify two types of tasks: finding the bin to which an element of data belongs and incrementing the appropriate entry in bin counts.
- For the second step, there must be a communication between the computation of the appropriate bin and incrementing an element of bin counts.

If we represent our tasks with ovals and communications with arrows, we'll get a diagram that looks something like that shown in Figure 2.21.

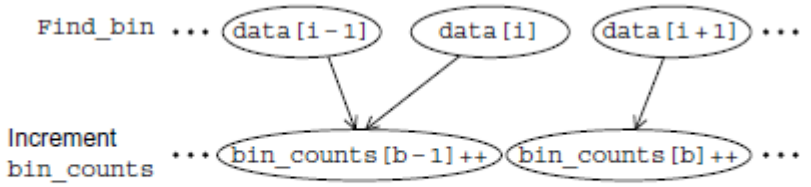


FIGURE 2.21

The first two stages of Foster's methodology

- ✓ Here, the task labelled with "data[i]"is determining which bin the value data[i] belongs to, and the task labelled with "bin counts[b]++" is incrementing bin counts[b].
- ✓ For any fixed element of data, the tasks "find the bin b for element of data" and "increment bin counts[b]" can be aggregated, since the second can only happen once the first has been completed.
- ✓ However, when we proceed to the final or mapping step, we see that if two processesor threads are assigned elements of data that belong to the same bin b, they'll both result in execution of the statement bin_counts[b]++.
- ✓ If bin_counts[b] is shared (e.g., the array bin counts is stored in shared-memory), then this will result in a race condition.

- ✓ If bin_counts has been partitioned among the processes/threads, then updates to its elements will require communication.
- ✓ An alternative is to store multiple "local" copies of bin counts and add the values in the local copies after all the calls to Find_bin.
- ✓ If the number of bins, bin count, isn't absolutely gigantic, there shouldn't be a problem with this.

So let's pursue this alternative, since it is suitable for use on both shared- and distributed-memory systems.In this setting, we need to update our diagram so that the second collection of tasks increments loc_bin_cts[b]. We also need to add a third collection of tasks,adding the various loc_bin_cts[b] to get_bin_counts[b]

Now if we create an array loc bin cts for each process/thread, then we can map the tasks in the first two groups as follows:

- ✓ Elements of data are assigned to the processes/threads so that each process/thread
  gets roughly the same number of elements.
- ✓ Each process/thread is responsible for updating its loc bin cts array on the basis
  of its assigned elements.

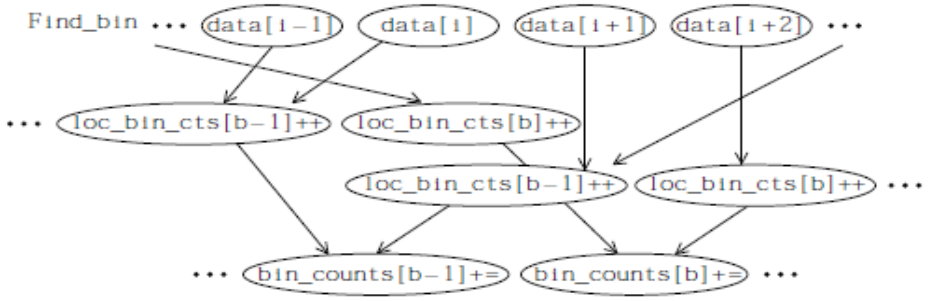


FIGURE 2.22

Alternative definition of tasks and communication

- ✓ To finish up, we need to add the elements loc_bin_cts[b] into bin_counts[b].
- ✓ If both the number of processes/threads is small and the number of bins is small, all
  of the additions can be assigned to a single process/thread.
- ✓ If the number of bins is much larger than the number of processes/threads, we can divide the bins among the processes/threads in much the same way that we divided the elements of data.
- ✓ The only difference is that now the sending process threads are sending an array, and the receiving process/threads are receiving and adding an array.
- ✓ Figure 2.23 shows an example with eight processes/threads. Each circle in the top row corresponds to a process/thread.
- ✓ Between the first and the second rows, the odd-numbered processes/threads make their loc_bin_cts available to the even-numbered processes/threads. Then in thesecond row, the even-numbered processes/threads add the new counts to their existing counts.
- ✓ Between the second and third rows the process is repeated with the processes/threads whose ranks aren't divisible by four sending to those whose are. This process repeated until process/thread 0 has computed bin counts.

Performance – Scalability – Synchronization and data sharing – Data races – Synchronization primitives (mutexes, locks, semaphores, barriers) – deadlocks and livelocks – communication between threads (condition variables, signals, message queues and pipes).

Performance

Algorithm complexity is important

Algorithmic complexity represents the expected performance of a section of code as the number of elements being processed increases. In the limit, the code with the greatest algorithmic complexity will dominate the runtime of the application.

Assume that your application has two regions of code, one that is O(N) and another that is O(N2). If you run a test workload of 100 elements, you may find that the O(N) code takes longer to execute, because there may be more instructions associated with the computation on each element. However, if you were to run a workload of 10,000 elements, then the more complex routine would start to show up as important, assuming it did not completely dominate the runtime of the application.

Picking a small workload will mislead you as to which parts of the code need to be optimized. You may have spent time optimizing the algorithmically simpler part of the code, when the performance of the application in a real-world situation will be dominated by the algorithmically complex part of the code. This emphasizes why it is important to select appropriate workloads for developing and testing the application. Different parts of the application will scale differently as the workload size changes, and regions that appear to take no time can suddenly become dominant.

Another important point to realize is that a change of algorithm is one of the few things that can make an order of magnitude difference to performance. If 80% of the application's runtime was spent sorting a 1,000-element array, then switching from a bubble sort to a quicksort could make a 300× difference to the performance of that function, making the time spent sorting 300× smaller than it previously was. The 80% of the runtime spent sorting would largely disappear, and the application would end up running about five times faster.

Table 2.1 shows the completion time of a task with different algorithmic complexities as the number of elements grows. It is assumed that the time to complete a single unit of work is 100ns. As the table illustrates, it takes remarkably few elements for an O(N2) algorithm to start consuming significant amounts of time.

Table 2.1 **Execution Duration at Different Algorithm Complexities**

Elements	O(1)	O(N)	O(N $\log_2$ N)	O(N^2)
1	100ns	100ns	100ns	100ns
10	100ns	1,000ns	3,322ns	10,000ns
100	100ns	10,000ns	66,439ns	1,000,000ns
1,000	100ns	100,000ns	996,578ns	100,000,000ns
10,000	100ns	1,000,000ns	13,287,712ns	10,000,000,000ns

The same information can be presented as a chart of runtimes versus the number of elements. Figure 2.1 makes the same point rather more dramatically. It quickly becomes apparent that the runtime for an O(N2) algorithm will be far greater than one that is linear or logarithmic with respect to the number of elements.

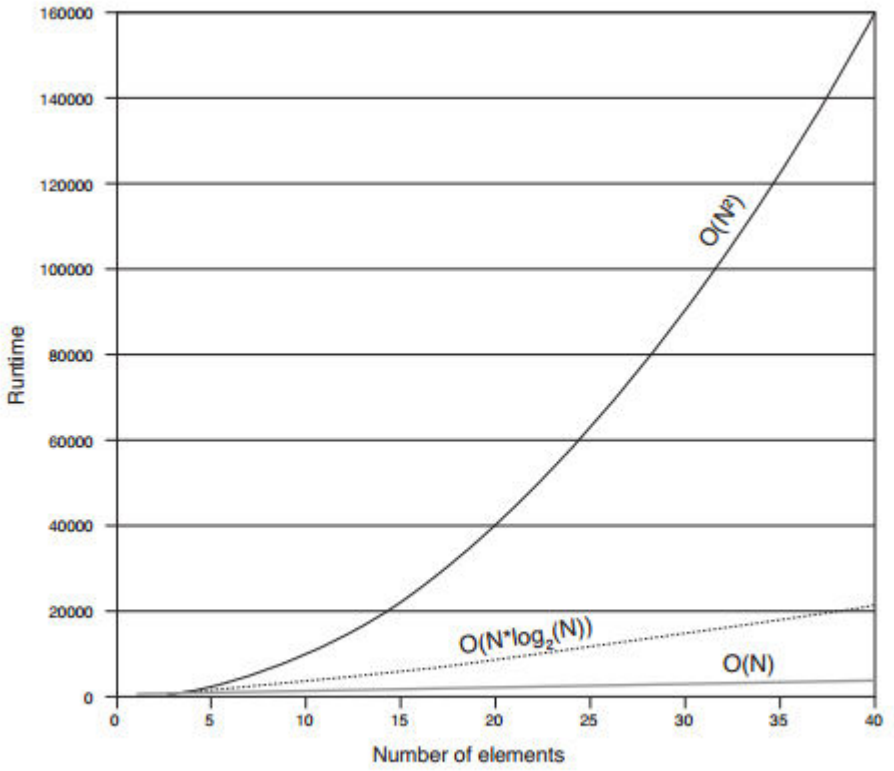


Figure 2.1 Different orders of algorithmic complexity

Structure Impacts Performance

Three attributes of the construction of an application can be considered as "structure."

- The first of these is the build structure, such as how the source code is distributed between the source files.
- The second structure is how the source files are combined into applications and supporting libraries.
- Finally, and probably the most obvious, is that way data is organized in the application

Performance and Convenience Trade-Offs in Source Code and Build Structures

The structure of the source code for an application can cause differences to its performance.

Source code is often distributed across source files for the convenience of the developers.

Performance opportunities are lost when the compiler sees only a single file at a time. The single file may not present the compiler with all the opportunities for optimizations that it might have had if it were to see more of the source code. This kind of limitation is visible when a program uses an accessor function—a short

function that returns the value of some variable. A trivial optimization is for the compiler to replace this function call with a direct load of the value of the variable.

Listing 2.5 **Accessor Functions**

```
#include <stdio.h>

int a;

void setvalue( int v ) { a = v; }

int  getvalue() { return a; }

void main()
{
  setvalue( 3 );
  printf( "The value of a is %i\n", getvalue() );
}
```

The code in Listing 2.5 can be replaced with the equivalent but faster code shown in Listing 2.6. This is an example of inlining within a source file. The calls to the routines getvalue() and setvalue() are replaced by the actual code from the functions.

Listing 2.6 **Pseudosource Code After Inlining Optimization**

```
#include <stdio.h>

int a;

void main()
{
  a = 3;
  printf( "The value of a is %i\n", a );
}
```

One common approach to building is to use either static or archive libraries as part of the build process. These libraries combine a number of object files into a single library, and at link time, the linker extracts the relevant code from the library. Listing 2.7 shows the steps in this process. In this case, two source files are combined into a single archive, and that archive is used to produce the application.

Listing 2.7 **Creating an Archive Library**

```
$ cc -c a.c
$ cc -c b.c
$ ar -r lib.a a.o b.o
ar: creating lib.a
$ cc main.c lib.a
```

There are three common reasons for using static libraries as part of the build process: n For “aesthetic” purposes, in that the final linking of the application requires fewer objects. The build process appears to be cleaner because many individual object files are combined into static libraries, and the smaller set appears on the link line. The libraries might also represent bundles of functionality provided to the executable. n To produce a similar build process whether the application is built to use static or dynamic libraries. Each library can be provided as either a static or a dynamic version, and it is up to the developer to decide which they will use. This is common when the library is distributed as a product for developers to use. n To hide build issues, but this is the least satisfactory reason. For example, an archive library can contain multiple versions of the same routine. At link time, the linker will extract the first version of this routine that it encounters, but it will not warn that there are

multiple versions present. If the same code was linked using individual object files without having first combined the object files into an archive, then the linker would fail to link the executable.

Listing 2.8 **Example of a Static Library Hiding Build Issues**

```
$ more a.c
#include <stdio.h>
void status()
{
  printf("In status of A\n");
}

$ more b.c
#include <stdio.h>
void status()
{
   printf("In status of B\n");
}

$ cc -c a.c
$ cc -c b.c
$ ar -r lib.a a.o b.o
ar: creating lib.a

$ more main.c
void status();
void main()
{
 status();
}

$ cc main.c lib.a
$ a.out
In status of A

$ cc main.c a.o b.o
ld: fatal: symbol 'status' is multiply-defined:
        (file a.o type=FUNC; file b.o type=FUNC);
ld: fatal: File processing errors. No output written to a.out
```

Using Libraries to Structure Applications

Libraries are the usual mechanism for structuring applications as they become larger. There are some good technical reasons to use libraries:

- Common functionality can be extracted into a library that can be shared between different projects or applications. This can lead to better code reuse, more efficient use of developer time, and more effective use of memory and disk space.
- Placing functionality into libraries can lead to more convenient upgrades where only the library is upgraded instead of replacing all the executables that use the library.
- Libraries can provide better separation between interface and implementation. The implementation details of the library can be hidden from the users, allowing the implementation of the library to evolve while maintaining a consistent interface.
- Stratifying functionality into libraries according to frequency of use can improve application start-up time and memory footprint by loading only the

libraries that are needed. Functionality can be loaded on demand rather than setting up all possible features when the application starts.

- Libraries can be used as a mechanism to dynamically provide enhanced functionality. The functionality can be made available without having to change or even restart the application.
- Libraries can enable functionality to be selected based on the runtime environment or characteristics of the system. For instance, an application may load different optimized libraries depending on the underlying hardware or select libraries at runtime depending on the type of work it is being asked to perform.

On the other hand, there are some nontechnical reasons why functionality gets placed into libraries. These reasons may represent the wrong choice for the user.

- Libraries often represent a convenient product for an organizational unit. One group of developers might be responsible for a particular library of code, but that does not automatically imply that a single library represents the best way for that code to be delivered to the end users.
- Libraries are also used to group related functionality. For example, an application might contain a library of string-handling functions. Such a library might be appropriate if it contains a large body of code. On the other hand, if it contains only a few small routines, it might be more appropriate to combine it with another library.

There are a few contributors to cost:

- Library calls may be implemented using a table of function addresses. This table may be a list of addresses for the routines included in a library. A library routine calls into this table, which then jumps to the actual code for the routine.
- Each library and its data are typically placed onto new TLB entries. Calls into a library will usually also result in an ITLB miss and possibly a DTLB miss if the code accesses library-specific data.
- If the library is being lazy loaded (that is, loaded into memory on demand), there will be costs associated with disk access and setting up the addresses of the library functions in memory.
- Unix platforms typically provide libraries as position-independent code. This enables the same library to be shared in memory between multiple running applications. The cost of this is an increase in code length. Windows makes the opposite trade-off; it uses position-dependent code in libraries, reducing the opportunity of sharing libraries between running applications but producing slightly faster code.

Listing 2.9 **Defining Two Libraries**

```
$ more liba.c
#include <stdio.h>

void ina()
{
  printf( "In library A\n" );
}

$ more libb.c
#include <stdio.h>

void inb()
{
   printf( "In library B\n" );
}
```

Impact of Data Structures on Performance

When an application needs an item of data, it fetches it from memory and installs it in cache.

The idea with caches is that data that is frequently accessed will become resident in the cache. The cost of fetching data from the cache is substantially lower than the cost of fetching it from memory. Hence, the application will spend less time waiting for
frequently accessed data to be retrieved from memory.

The amount of data loaded into each level of cache by a load instruction depends on the size of the cache line. 64 bytes is a typical length for a cache line; however, some caches have longer lines than this, and some caches have shorter lines. Often the caches that are closer to the processor have shorter lines, and the lines further from the processor have longer lines.

Figure 2.2 illustrates what happens when a line is fetched into cache from memory.

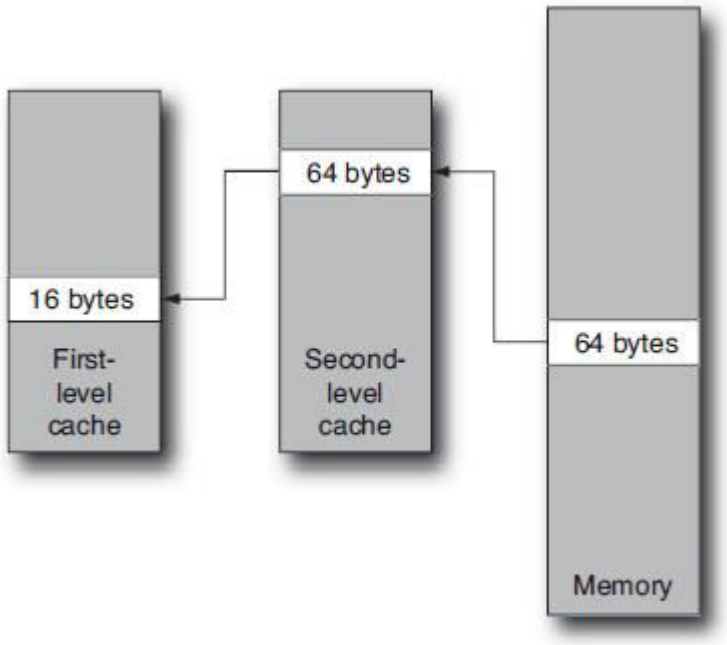


Figure 2.2 Fetching data from memory into caches

On a cache miss, a cache line will be fetched from memory and installed into the second level cache. The portion of the cache line requested by the memory operation is installed into the first-level cache.
In this scenario,

- Accesses to data on the same 16-byte cache line as the original item will also be available from the first-level cache.
- Accesses to data that share the same 64-byte cache line will be fetched from the second-level cache.
- Accesses to data outside the 64-byte cache line will result in another fetch from memory.

If data is fetched from memory when it is needed, the processor will experience the entire latency of the memory operation. On a modern processor, the time taken to perform this fetch can be several hundred cycles.
There are techniques that reduce this latency:

- *Out-of-order execution* is where the processor will search the instruction stream for future instructions that it can execute.
- *Hardware prefetching* of data streams Hardware prefetching can be very effective in situations where data is fetched as a stream or through a strided access pattern. It is not able to prefetch data where the access pattern is less apparent.
- *Software prefetching* is the act of adding instructions to fetch data from memory before it is needed.

Another approach to covering memory latency costs is with CMT processors. When one thread stalls because of a cache miss, the other running threads get to use the processor resources of the stalled thread. This approach does not improve the execution speed of a single thread. This can enable the processor to achieve more work by sustaining more active threads, improving throughput rather than single-threaded performance.
There are a number of common coding styles that can often result in suboptimal layout
of data in memory.
Improving Performance Through Data Density and Locality
Paying attention to the order in which variables are declared and laid out in memory can improve performance. When a load brings a variable in from memory, it also fetches the rest of the cache line in which the variable resides. Placing variables that are commonly accessed together into a structure so that they reside on the same cache line will lead to performance gains.
Data Structure

```
struct s
{
int var1;
int padding1[15];
int var2;
int padding2[15];
```

}
When the structure member var1 is accessed, the fetch will also bring in the surrounding
64 bytes. The size of an integer variable is 4 bytes, so the total size of var1 plus padding1 is 64 bytes. This ensures that the variable var2 is located on the next cache line.
Important Structure Members Are Likely to Share a Cache Line

```
struct s
{
int var1;
int var2;
int padding1[15];
int padding2[15];
}
```

If the structure does not fit exactly into the length of the cache line, there will be situations when the adjacent var1 and var2 are split over two cache lines. This introduces a dilemma. Is it better to pack the structures as close as possible to fit as many of them as possible into the same cache line, or is it better to add padding to the structures to make them consistently align with the cache line boundaries? Figure 2.3 shows the two situations.
The answer will depend on various factors. In most cases, the best answer is probably to pack the structures as tightly as possible. This will mean that when one structure is accessed, the access will also fetch parts of the surrounding structures. The situation

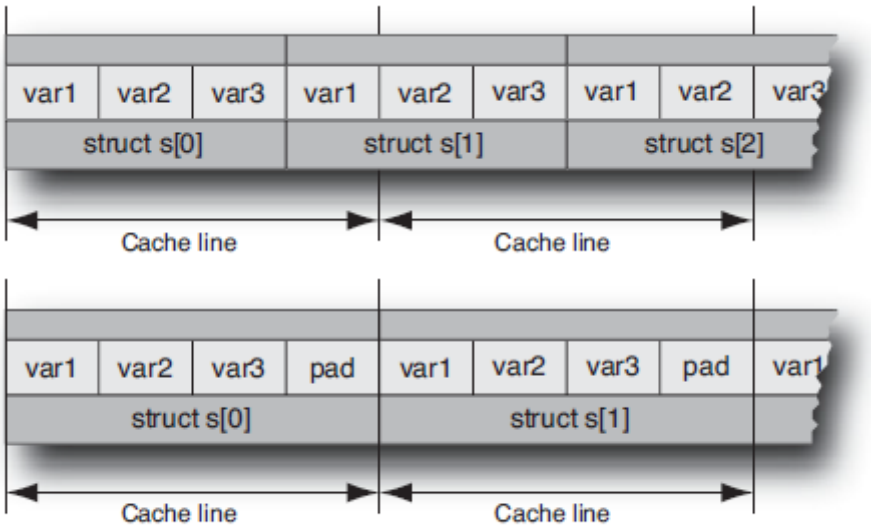


Figure 2.3 Using padding to align structures

Where it is appropriate to add padding to the structure is when the structures are always accessed randomly, so it is more important to ensure that the critical data is not split across a cache line.
The performance impact of poorly ordered structures can be hard to detect. The cost is spread over all the accesses to the structure over the entire application. Reordering the structure members can improve the performance for all the routines that access

the structures. Determining the optimal layout for the structure members can also be difficult.

One guideline would be to order the structure members by access frequency or group them by those that are accessed in the hot regions of code. It is also worth considering that changing the order of structure members could introduce a performance regression if the existing ordering happens to have been optimal for a different frequently executed region of code.

A similar optimization is structure splitting, where an existing structure is split into members that are accessed frequently and members that are accessed infrequently. If the infrequently accessed structure members are removed and placed into another structure, then each fetch of data from memory will result in more of the critical structures being fetched in one action. Taking the previous example, where we assume that var3 is rarely needed, we would end up with a resulting pair of structures, as shown in Figure 2.4.

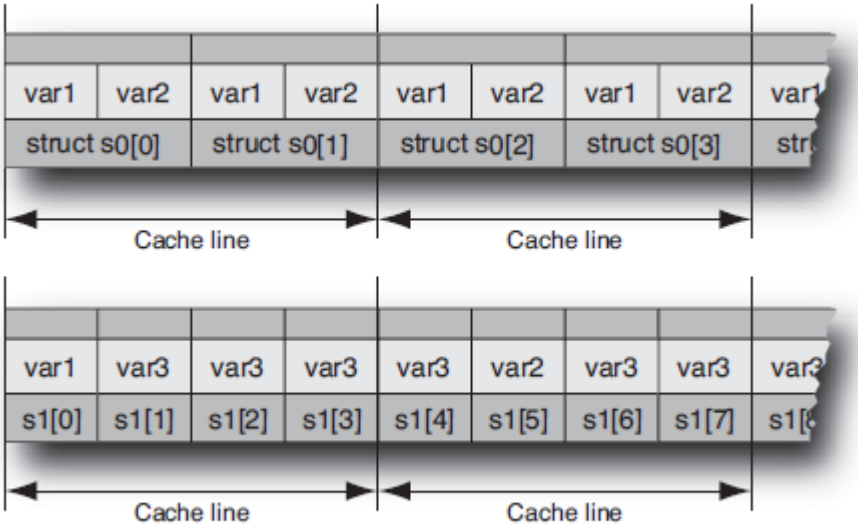


Figure 2.4 Using structure splitting to improve memory locality

In this instance, the original structure s has been split into two, with s0 containing all the frequently accessed data and s1 containing all the infrequently accessed data. In the limit, this optimization is converting what might be an array of structures into a set of arrays, one for each of the original structure members.

Selecting the Appropriate Array Access Pattern

One common data access pattern is striding through elements of an array. The performance of the application would be better if the array could be arranged so that the selected elements were contiguous.

shows an example of code accessing an array with a stride.

Noncontiguous Memory Access Pattern

```
{
double ** array;
double total=0;
```

```
…
for (int i=0; i<cols; i++)
for (int j=0; j<rows; j++)
total += array[j][i];
…
}
```

C/C++ arrays are laid out in memory so that the adjacent elements of the final index (in this case indexed by the variable i) are adjacent in memory; this is called *row-major* order. However, the inner loop within the loop nest is striding over the first index into the matrix and accessing the ith element of that array. These elements will not be located in contiguous memory.

In Fortran, the opposite ordering is followed, so adjacent elements of the first index are adjacent in memory. This is called *column-major* order. Accessing elements by a stride is a common error in codes translated from Fortran into C.

shows how memory is addressed in C, where adjacent elements in a row are adjacent in memory.

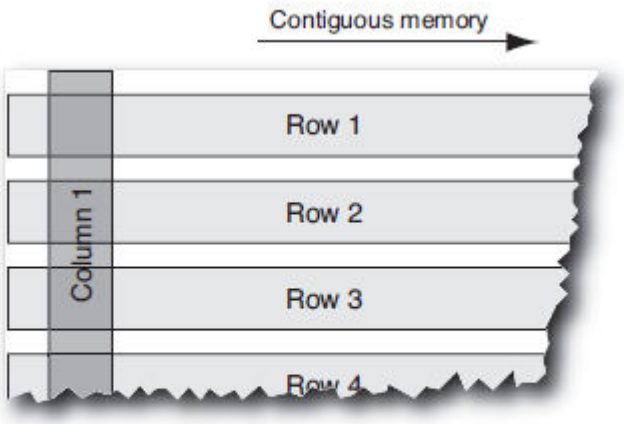


Figure 2.5 Row major memory ordering

Fortunately, most compilers are often able to correctly interchange the loops and improve the memory access patterns. However, there are many situations where the compiler is unable to make the necessary transformations because of aliasing or the order in which the elements are accessed in the loop. In these cases, it is necessary for the developer to determine the appropriate layout and then restructure the code appropriately.

Choosing Appropriate Data Structures

Choosing the best structure to hold data, such as choosing an algorithm of the appropriate complexity, can have a major impact on overall performance. This harks back to the discussions of algorithmic complexity earlier in this chapter. Some structures will be efficient when data is accessed in one pattern, while other structures will be more efficient if the access pattern is changed.

Consider a simple example. Suppose you have a dictionary of words for a spellchecker

application. You don't know at compile time how many words will be in the dictionary, so the easiest way to cope with this might be to read in the words and place
them onto a linked list, as shown in Figure 2.6.

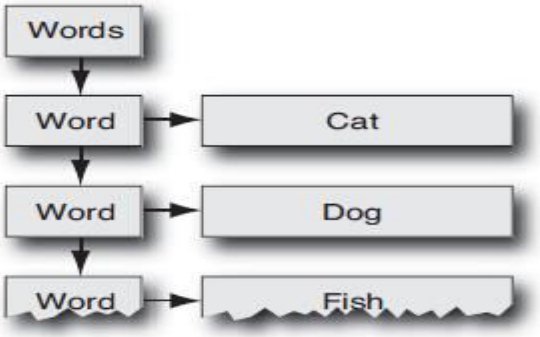


Figure 2.6 Using a linked list to hold an ordered list of words

Every time the application needs to check whether a word is in the dictionary, it traverses the linked list of words, so a spell-check of the entire document is an O(N2) activity.
An alternative implementation might be to allocate an array of known length to hold pointers to the various words, as shown in Figure 2.7.

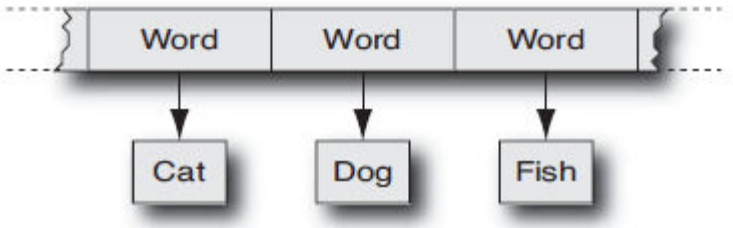


Figure 2.7 Using an array to hold an ordered list of words

Although there might be some complications in getting the array to be the right length to hold all the elements, the benefit comes from being able to do a binary search
on the sorted list of words held in the array. A binary search is an O(log2(N)) activity, so
performing a spell-check on an entire document would be an O(N∗log2(N)) activity, which, as indicated earlier, would be a significantly faster approach.
As in any example, there are undoubtedly better structures to choose for holding a dictionary of words. Choosing the appropriate one for a particular application is a case of
balancing the following factors:
Programmer time to implement the algorithm. There will probably be constraints on the amount of time that a developer can spend on implementing a single part of the application.
User sensitivity to application performance. Some features are rarely used, so a user might accept that, for example, performing a spell-check on an entire document will take time. It may also be the case that the compute part of the task is not time critical; in the case of a spell-check, if a spelling error is reported, the user may

spend time reading the text to determine the appropriate word to use, during which the application could continue and complete the spell-check of the rest of the document.
The problem size is not large enough to justify the more complex algorithm. If the application is limited to documents of only a few hundred words, it is unlikely that a spell-check of the entire document would ever take more than about half a second. Any performance gains from the use of an improved algorithm would be unnoticeable.
In many situations, there are preexisting libraries of code that implement different data management structures. For C++, the Standard Template Library provides a wealth
of data structures. Careful coding to encapsulate the use of the data structures can minimize
developer time by allowing the original structures to be easily replaced with more efficient ones should that prove necessary.

Scalability

Data Races and the Scaling Limitations of Mutex Locks

Bugs because of data races are the most obvious manifestation of a parallelization problem. These relate to updates of variables without ensuring exclusive access to the variables. These are usually resolved by adding synchronization primitives (such as mutex locks) into the code to ensure exclusive access to the variables. Although mutex locks can be used to ensure that only a single thread has access to a resource at a time, they cannot enforce the ordering of accesses to data.
An alternative approach is required if there is an ordering constraint on the accesses to shared resources. For example, if two threads need to update a variable, a mutex can ensure that they do not update the variable at the same time. However, a mutex cannot force one of the two threads to be the last to perform the update. The problem with adding mutex locks into the code is that they serialize the access to the variables. Only a single thread can hold the lock, so if there are multiple threads that need to access the data, the application effectively runs serially because only one thread can make progress at a time. Even if requiring the mutex lock is a rare event, it can become a bottleneck if the lock is held for a long time or if there are many threads requiring access to the lock.

Scaling of library code.

Here we find out scaling issues in code provided not only in application and also in libraries.

Code to Testing Scaling of malloc() and free()

```
#include <stdlib.h>
#include <pthread.h>
int nthreads;
void *work( void * param )
{
int count = 1000000 / nthreads;
```

```
for( int i=0; i<count; i++ )
{
void *mem = malloc(1024);
free( mem );
}
}
int main( int argc, char*argv[] )
{
pthread_t thread[50];
nthreads = 8;
if ( argc > 1 ) { nthreads = atoi( argv[1] ); }
for( int i=0; i<nthreads; i++ )
{
pthread_create( &thread[i], 0, work, 0 );
}
for( int i=0; i<nthreads; i++ )
{
pthread_join( thread[i], 0 );
}
return 0;
}
```

- If a default implementation of malloc() and free() uses a single mutex lock then performance will not improve with multiple threads.
- Consider an alternative malloc() that uses a different algorithm. Each thread has its own heap of memory, so it does not require a mutex lock. This alternative malloc() scales as the number of threads increases.
- As expected, the default implementation does not scale, so the runtime does not improve. The increase in runtime is because of more threads contending for the single mutex lock.
- The alternative implementation shows very good scaling. As the number of threads increases, the runtime of the application decreases.
- For the singlethreaded case, the default malloc() provides better performance than the alternative implementation. The algorithm that provides improved scaling also adds a cost to the single-threaded situation; it can be hard to produce an algorithm that is fast for the single- threaded case and scales well with multiple threads.

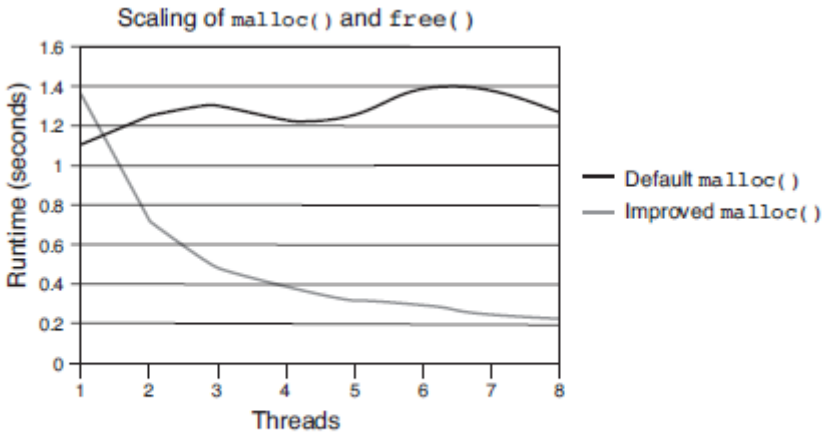


Figure 9.5 Scaling of two different implementations of `malloc()` and `free()`

Superlinear Scaling

Doubling the resources yet get more than double the performance as a result.

Example:

In most instances, going from one thread to two will result in, at most, a doubling of performance. However, there will be applications that do see super linear scaling—the application ends up running more than twice as fast. This is typically because the data that the application uses becomes cache resident at some point. Imagine an application that uses 4MB of data. On a processor with a 2MB cache, only half the data will be resident in the cache. Adding a second processor adds an additional 2MB of cache; then all the data becomes cache resident, and the time spent waiting on memory becomes substantially lower.

This program uses 64MB of memory.

```
#include <math.h>
#include <stdlib.h>
void func1( double*array, int n )
{
for( int i=1; i<n; i++ )
{
array[i] += array[i-1];
}
}
void func2( double *array,int n )
{
#pragma omp parallel for
for( int i=0; i<n; i++ )
{
array[i] = sin(array[i]);
}
}
int main()
{
double * array = calloc( sizeof(double), 1024*1024 );
for ( int i=0; i<100; i++ )
```

```
{
func1( array, 1024*1024 );
func2( array, 1024*1024 );
}
return 0;
}
```

Program with 64MB Memory Footprint

```
#include <stdlib.h>
double func1( double*array, int n )
{
double total = 0.0;
#pragma omp parallel for reduction(+:total)
for( int i=1; i<n; i++ )
{
total += array[i^29450];
}
return total;
}
int main()
{
double * array = calloc( sizeof(double), 8192*1024 );
for( int i=0; i<100; i++ )
{
func1( array, 8192*1024 );
}
}
```

- When the program is run on a single processor with 32MB of second-level cache, the
- program takes about 25 seconds to complete.
- When run using two threads on the same processor, the code completes in about 12 seconds and takes 25 seconds of user time.

This is the anticipated performance gain from using multiple threads. The code takes half the time but does the same amount of work. However, when run using two threads, with each thread bound to a separate processor, the program runs in just over four seconds of wall time, taking only eight seconds of user time.

Adding the second processor has increased the amount of cache available to the program, causing it to become cache resident. The data in cache has lower access latency, so the program runs significantly faster.

It is a different situation on a multicore processor. Adding an additional thread, particularly if it resides on the same core, does not substantially increase the amount of cache available to the program. So, a multicore processor is unlikely to see super linear speedup.

Hardware constraints applicable to improve scaling.

There are three critical areas that can make a large difference to scaling.

- The amount of bandwidth to cache and the memory will be divided among the active threads on the system.
- The design of the caches will determine how much time is lost because of capacity and conflict-induced cache misses.
- The way that the processor core pipelines are shared between active software threads will determine how instruction issue rates change as the number of active threads increases.

Bandwidth Sharing Between Cores

Bandwidth is another resource shared between threads. The bandwidth capacity of a system depends on the design of the processor and the memory system as well as the memory chips and their location in the system.

The bandwidth a processor can consume is a function of the number of outstanding memory requests and the rate at which these can be returned. These memory requests

can come from either hardware or software prefetches, as well as from load or store operations. Since each thread can issue memory requests, the more threads that a processor can run, the more bandwidth the processor can consume.

string-handling library routines such as strlen() or memset() can be large consumers of memory bandwidth.

Using memset to Measure Memory Bandwidth

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <pthread.h>
#include <sys/time.h>
#define BLOCKSIZE 1024*1025
int nthreads = 8;
char * memory;
double now()
{
struct timeval time;
gettimeofday( &time, 0 );
return (double)time.tv_sec + (double)time.tv_usec / 1000000.0;
}
void *experiment( void *id )
{
unsigned int seed = 0;
int count = 20000;
for( int i=0; i<count; i++ )
{
memset( &memory[BLOCKSIZE * (int)id], 0, BLOCKSIZE );
}
if ( seed == 1 ){ printf( "" ); }
}
int main( int argc, char* argv[] )
{
```

```
pthread_t threads[64];
memory = (char*)malloc( 64*BLOCKSIZE );
if ( argc > 1 ) { nthreads = atoi( argv[1] ); }
double start = now();
for( int i=0; i<nthreads; i++ )
{
pthread_create( &threads[i], 0, experiment, (void*)i );
}
for ( int i=0; i<nthreads; i++ )
{
pthread_join( threads[i], 0 );
}
double end = now();
printf( "%i Threads Time %f s Bandwidth %f GB/s\n", nthreads,
(end – start) ,
( (double)nthreads * BLOCKSIZE * 20000.0 ) /
( end – start) / 1000000000.0 );
return 0;
}
```

the bandwidth measured by the test code for one to eight virtual CPUs on a system with 64 virtual CPUs.

Memory Bandwidth Measured on a System with 64 Virtual CPUs

1 Threads Time 7.082376 s Bandwidth 2.76 GB/s
2 Threads Time 7.082576 s Bandwidth 5.52 GB/s
3 Threads Time 7.059594 s Bandwidth 8.31 GB/s
4 Threads Time 7.181156 s Bandwidth 10.89 GB/s
5 Threads Time 7.640440 s Bandwidth 12.79 GB/s
6 Threads Time 11.252412 s Bandwidth 10.42 GB/s
7 Threads Time 14.723671 s Bandwidth 9.29 GB/s
8 Threads Time 17.267288 s Bandwidth 9.06 GB/s

For this particular system, the bandwidth scales nearly linearly with the number of threads until about six threads. After six threads, the bandwidth reduces.

There are several effects that can cause this :

- The threads are interfering on the processor.
- A second interaction effect is if the threads start interfering in the caches, such as multiple threads attempting to load data to the same set of cache lines.
- One other effect is the behaviour of memory chips when they become saturated. the chips start experiencing queuing latencies where the response time for each request increases. Memory chips are arranged in banks. Accessing a particular address will lead to a request to a particular bank of memory. Each bank needs a gap between returning two responses. If multiple threads happen to hit the same bank, then the response time becomes governed by the rate at which the bank can return memory.

Memory Bandwidth Measured on a System with Four Virtual CPUs

1 Threads Time 7.437563 s Bandwidth 2.63 GB/s
2 Threads Time 15.238317 s Bandwidth 2.57 GB/s

3 Threads Time 24.580981 s Bandwidth 2.39 GB/s
4 Threads Time 37.457352 s Bandwidth 2.09 GB/s

False Sharing

False sharing is the situation where multiple threads are accessing items of data held on a single cache line.

Although the threads are all using separate items of data, the cache line itself is shared between them so only a single thread can write to it at any one time. This is purely a performance issue

Example of False Sharing

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/time.h>
double now()
{
struct timeval time;
gettimeofday( &time, 0 );
return (double)time.tv_sec + (double)time.tv_usec / 1000000.0;
}
#define COUNT 100000000
volatile int go = 0;
volatile int counters[20];
void *spin( void *id )
{
int myid = (int)id + 1;
while( !go ) {}
counters[myid] = 0;
while ( counters[myid]++ < COUNT ) {}
}
int main( int argc, char* argv[] )
{
pthread_t threads[256];
int nthreads = 1;
if ( argc > 1 ) { nthreads = atoi( argv[1] ); }
for( int i=1; i<nthreads; i++ )
{
pthread_create( &threads[i], 0, spin, (void*)i );
}
double start = now();
go = 1;
spin( 0 );
double end = now();
printf("Time %f ns\n", ( end – start ) );
for( int i=0; i<nthreads; i++ )
{
pthread_join( threads[i], 0 );
}
```

```
return 0;
}
```
If we run ABOVE CODE WITH a single thread, the thread completes its work in about nine seconds on a system with two dual-core processors. Using four threads on the same system results in a runtime for the code of about 100 seconds—a slowdown of about 10 times.

It is very easy to solve false sharing by padding the accessed structures so that the variable used by each thread resides on a separate cache line.

Data Padded to Avoid False Sharing

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/time.h>
double now()
{
struct timeval time;
gettimeofday( &time, 0 );
return (double)time.tv_sec + (double)time.tv_usec / 1000000.0;
}
#define COUNT 100000000
volatile int go = 0;
volatile int counters[320];
void *spin( void *id )
{
int myid = ( (int)id + 1) * 16;
while( !go ) {}
counters[myid] = 0;
while ( counters[myid]++ < COUNT ) {}
}
int main( int argc, char* argv[] )
{
pthread_t threads[256];
int nthreads = 1;
if ( argc > 1 ) { nthreads = atoi( argv[1] ); }
nthreads--;
for( int i=1; i<nthreads+1; i++ )
{
pthread_create( &threads[i], 0, spin, (void*)i );
}
double start = now();
go=1;
spin( 0 );
double end = now();
printf( "Time %f s\n", ( end – start ) );
for( int i=0; i<nthreads; i++ )
{
pthread_join( threads[i], 0 );
}
```

```
return 0;
}
```
The modified code takes about nine seconds to run with four threads on the same machine. false sharing will turn up as an elevated number of cache misses on a particular memory operation, and it is hard to distinguish this from the normal cache misses that occur in all applications.

Cache Conflict and Capacity

The two issues that can occur with shared caches: capacity misses and conflict misses.

A conflict cache miss is where one thread has caused data needed by another thread to be evicted from the cache. Data structures such as stacks tend to be aligned on cache line boundaries, which
increases the likelihood that structures from different processes will map onto the same
address.

Code to Print the Stack Address for Different Threads

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
pthread_barrier_t barrier;
void* threadcode( void* param )
{
int stack;
printf("Stack base address = %x for thread %i\n", &stack, (int)param);
pthread_barrier_wait( &barrier );
}
int main( int argc, char*argv[] )
{
pthread_t threads[20];
int nthreads = 8;
if ( argc > 1 ) { nthreads = atoi( argv[1] ); }
pthread_barrier_init( &barrier, 0, nthreads );
for( int i=0; i<nthreads; i++ )
{
pthread_create( &threads[i], 0, threadcode, (void*)i );
}
for( int i=0; i<nthreads; i++ )
{
pthread_join( threads[i], 0 );
}
pthread_barrier_destroy( &barrier );
return 0;
}
```

The expected output when this code is run on 32-bit Solaris indicates that threads are created with a 1MB offset between the start of each stack. For a processor with a cache size that is a power of two and smaller than 1MB, a stride of 1MB would ensure the base of the stack for all threads is in the same set of cache lines. The associativity of the cache will reduce the chance that this would be a problem. A

cache with an associativity greater than the number of threads sharing is less likely to have a problem with conflict misses.

Data Races

Data races are the most common programming error found in parallel code. A data race occurs when multiple threads use the same data item and one or more of those threads are updating it. It is best illustrated by an example. Suppose you have the code shown in Listing 4.1, where a pointer to an integer variable is passed in and the function increments the value of this variable by 4.

Listing 4.1 **Updating the Value at an Address**

```
void update(int * a)
{
   *a = *a + 4;
}
```

Listing 4.2 **SPARC Disassembly for Incrementing a Variable Held in Memory**

```
ld  [%o0], %o1   // Load *a
add  %o1, 4, %o1 // Add 4
st   %o1, [%o0]  // Store *a
```

Table 4.1 **Two Threads Updating the Same Variable**

Value of variable a = 10	
Thread 1	**Thread 2**
`ld  [%o0], %o1  // Load %o1 = 10`	`ld  [%o0], %o1  // Load %o1 = 10`
`add %01, 4, %o1 // Add  %o1 = 14`	`add %01, 4, %o1 // Add %o1 = 14`
`st  %o1, [%o0]  // Store %o1`	`st  %o1, [%o0]  // Store %o1`
Value of variable a = 14	

In the example, each thread adds 4 to the variable, but because they do it at exactly the same time, the value 14 ends up being stored into the variable. If the two threads had executed the code at different times, then the variable would have ended up with the value of 18.

Another situation might be when one thread is running, but the other thread has been context switched off of the processor. Imagine that the first thread has loaded the value of the variable a and then gets context switched off the processor. When it eventually runs again, the value of the variable a will have changed, and the final store of the restored thread will cause the value of the variable a to regress to an old value

The tools used for detecting data races

Listing 4.3 **Code Containing Data Race**

```
#include <pthread.h>

int counter = 0;

void * func(void * params)
{
   counter++;
}

void main()
{
  pthread_t thread1, thread2;
  pthread_create( &thread1, 0, func, 0);
  pthread_create( &thread2, 0, func, 0);
  pthread_join( thread1, 0 );
  pthread_join( thread2, 0 );
}
```

"Using POSIX Threads." The code creates two threads, both of which execute the routine func(). The main thread then waits for both the child threads to complete their work. Both threads will attempt to increment the variable counter. We can compile this code with GNU gcc and then use Helgrind, which is part of the Valgrind1 suite, to identify the data race. Valgrind is a tool that enables an application to be instrumented and its runtime behavior examined. The Helgrind tool uses this instrumentation to gather data about data races. Listing 4.4 shows the output from Helgrind.

Listing 4.4 **Using Helgrind to Detect Data Races**

```
$ gcc -g race.c -lpthread
$ valgrind --tool=helgrind ./a.out
...
==4742==
==4742== Possible data race during write of size 4
         at 0x804a020 by thread #3
==4742==    at 0x8048482: func (race.c:7)
==4742==    by 0x402A89B: mythread_wrapper (hg_intercepts.c:194)
==4742==    by 0x40414FE: start_thread
            (in /lib/tls/i686/cmov/libpthread-2.9.so)
==4742==    by 0x413849D: clone (in /lib/tls/i686/cmov/libc-2.9.so)
==4742==  This conflicts with a previous write of size 4 by thread #2
==4742==    at 0x8048482: func (race.c:7)
==4742==    by 0x402A89B: mythread_wrapper (hg_intercepts.c:194)
==4742==    by 0x40414FE: start_thread
            (in /lib/tls/i686/cmov/libpthread-2.9.so)
==4742==    by 0x413849D: clone (in /lib/tls/i686/cmov/libc-2.9.so)
```

The output from Helgrind shows that there is a potential data race between two threads, both executing line 7 in the file race.c.

Another tool that is able to detect potential data races is the Thread Analyzer in Oracle Solaris Studio. This tool requires an instrumented build of the application, data collection is done by the collect tool, and the graphical interface is launched with the command tha. Listing 4.5 shows the steps to do this.

Listing 4.5 **Detecting Data Races Using the Sun Studio Thread Analyzer**

```
$ cc -g -xinstrument=datarace race.c
$ collect -r on ./a.out
Recording experiment tha.1.er ...
$ tha tha.1.er&
```

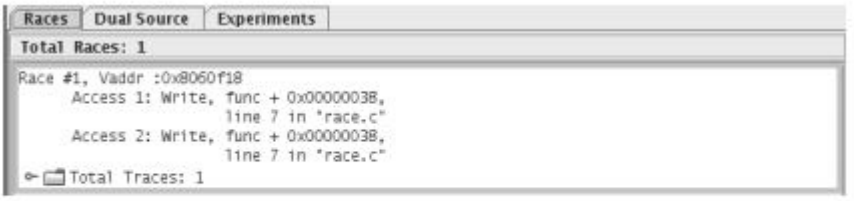


Figure 4.1 List of data races detected by the Solaris Studio Thread Analyzer

The initial screen of the tool displays a list of data races, as shown in Figure 4.1. Once the user has identified the data race they are interested in, they can view the source code for the two locations in the code where the problem occurs. In the example, shown in Figure 4.2, both threads are executing the same source line.

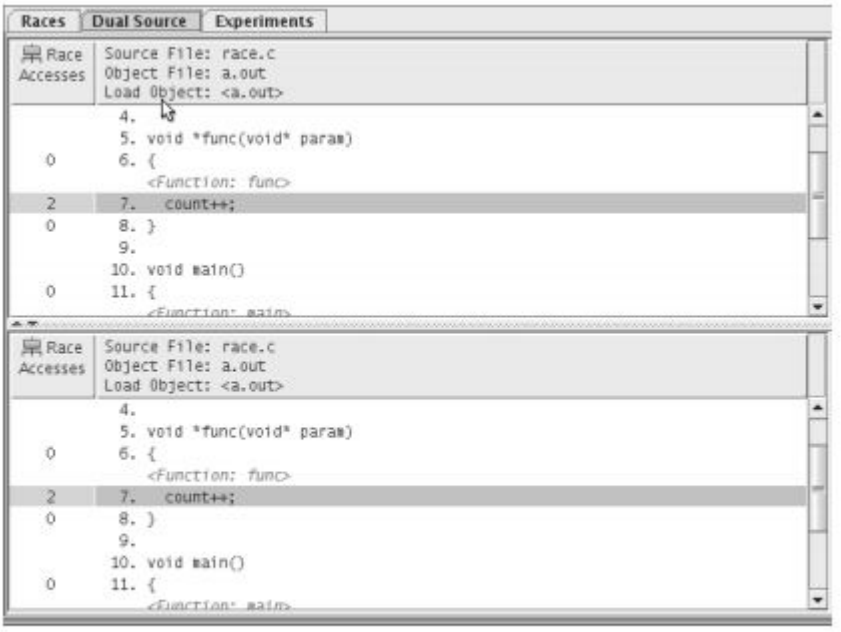


Figure 4.2 Source code with data race shown in Solaris Studio Thread Analyzer

Synchronization primitives:

Synchronization is used to coordinate the activity of multiple threads. Most operating systems provide a rich set of synchronization primitives. It is usually most appropriate to use these rather than attempting to write custom methods of synchronization.

the tools will be able to do a better job of detecting data races or correctly labeling synchronization costs.

Mutexes and Critical Regions

The simplest form of synchronization is a mutually exclusive (mutex) lock. Only one thread at a time can acquire a mutex lock, so they can be placed around a data structure to ensure that the data structure is modified by only one thread at a time. Listing 4.7 shows how a mutex lock could be used to protect access to a variable.

Listing 4.7 **Placing Mutex Locks Around Accesses to Variables**

```
int counter;

mutex_lock mutex;

void Increment()
{
   acquire( &mutex );
   counter++;
   release( &mutex );
}

void Decrement()
{
   acquire( &mutex );
   counter--;
   release( &mutex );
}
```

In the example, the two routines Increment() and Decrement() will either increment or decrement the variable counter. To modify the variable, a thread has to first acquire the mutex lock. Only one thread at a time can do this; all the other threads that want to acquire the lock need to wait until the thread holding the lock releases it.

Both routines use the same mutex; consequently, only one thread at a time can either increment or decrement the variable counter. If multiple threads are attempting to acquire the same mutex at the same time, then only one thread will succeed, and the other threads will have to wait. This situation is known as a contended mutex.

The region of code between the acquisition and release of a mutex lock is called a critical section, or critical region. Code in this region will be executed by only one thread at a time.

As an example of a critical section, imagine that an operating system does not have an implementation of malloc() that is thread-safe, or safe for multiple threads to call at the same time. One way to fix this is to place the call to malloc() in a critical section by surrounding it with a mutex lock, as shown in Listing 4.8.

Listing 4.8 **Placing a Mutex Lock Around a Region of Code**

```
void * threadSafeMalloc( size_t size )
{
   acquire( &mallocMutex );
   void * memory = malloc( size );
   release( &mallocMutex );
   return memory;
}
```

If all the calls to malloc() are replaced with the threadSafeMalloc() call, then only one thread at a time can be in the original malloc() code, and the calls to malloc() become

thread-safe. Threads block if they attempt to acquire a mutex lock that is already held by another thread.

Blocking means that the threads are sent to sleep either immediately or after a few unsuccessful attempts to acquire the mutex. One problem with this approach is that it can serialize a program.

If multiple threads simultaneously call threadSafeMalloc(), only one thread at a time will make progress. This causes the multithreaded program to have only a single executing thread, which stops the program from taking advantage of multiple cores

Spin Locks

Spin locks are essentially mutex locks. The difference between a mutex lock and a spin lock is that a thread waiting to acquire a spin lock will keep trying to acquire the lock without sleeping.

The advantage of using spin locks is that they will acquire the lock as soon as it is released, whereas a mutex lock will need to be woken by the operating system before it can get the lock.

The disadvantage is that a spin lock will spin on a virtual CPU monopolizing that resource. In comparison, a mutex lock will sleep and free the virtual CPU for another thread to use.

Semaphores

Semaphores are counters that can be either incremented or decremented. An example might be a buffer that has a fixed size. Every time an element is added to a buffer, the number of available positions is decreased. Every time an element is removed, the number available is increased.

Semaphores can also be used to mimic mutexes; if there is only one element in the semaphore, then it can be either acquired or available, exactly as a mutex can be either locked or unlocked.

Semaphores will also signal or wake up threads that are waiting on them to use available resources; hence, they can be used for signaling between threads. For example, a thread might set a semaphore once it has completed some initialization. Other threads could wait on the semaphore and be signaled to start work once the initialization is complete

Readers-Writer Locks

Data races are a concern only when shared data is modified. Multiple threads reading the shared data do not present a problem. Read-only data does not, therefore, need protection with some kind of lock. However, sometimes data that is typically read-only needs to be updated. A readerswriter lock (or multiple-reader lock) allows many threads to read the shared data but can then lock the readers threads out to allow one thread to acquire a writer lock to modify the data. A writer cannot acquire the write lock until all the readers have released their reader locks. For this reason, the locks tend to be biased toward writers; as soon as one is queued, the lock stops allowing further readers to enter. This action causes the number of

readers holding the lock to diminish and will eventually allow the writer to get exclusive access to the lock.

Listing 4.9 **Using a Readers-Writer Lock**

```
int readData( int cell1, int cell2 )
{
  acquireReaderLock( &lock );
  int result = data[cell1] + data[cell2];
  releaseReaderLock( &lock );
  return result;
}

void writeData( int cell1, int cell2, int value )
{
  acquireWriterLock( &lock );
  data[cell1] += value;
  data[cell2] -= value;
  releaseWriterLock( &lock );
}
```

Barriers

There are situations where a number of threads have to all complete their work before any of the threads can start on the next task.

For example, suppose a number of threads compute the values stored in a matrix. The variable total needs to be calculated using the values stored in the matrix. A barrier can be used to ensure that all the threads complete their computation of the matrix before the variable total is calculated. Listing 4.10 shows a situation using a barrier to separate the calculation of a variable from its use.

Listing 4.10 **Using a Barrier to Order Computation**

```
Compute_values_held_in_matrix();
Barrier();
total = Calculate_value_from_matrix();
```

The variable total can be computed only when all threads have reached the barrier. This avoids the situation where one of the threads is still completing its computations while the other threads start using the results of the calculations. Notice that another barrier could well be needed after the computation of the value for total if that value is then used in further calculations. Listing 4.11 shows this use of multiple barriers.

Listing 4.11 **Use of Multiple Barriers**

```
Compute_values_held_in_matrix();
Barrier();
total = Calculate_value_from_matrix();
Barrier();
Perform_next_calculation( total );
```

Deadlocks and Live locks.

The fundamental ways to share access to resources between threads :

- Deadlock
- Livelocks

The deadlock, where two or more threads cannot make progress because the resources that they need are held by the other threads. It is easiest to explain this with an example. Suppose two threads need to acquire mutex locks A and B to complete some task. If thread 1 has already acquired lock A and thread 2 has already acquired lock B, then A cannot make forward progress because it is waiting for lock B, and thread 2 cannot make progress because it is waiting for lock A. The two threads are deadlocked.

Listing 4.13 **Two Threads in a Deadlock**

Thread 1

```
void update1()
{
   acquire(A);
   acquire(B); <<< Thread 1
                   waits here
   variable1++;
   release(B);
   release(A);
}
```

Thread 2

```
void update2()
{
   acquire(B);
   acquire(A); <<< Thread 2
                   waits here
   variable1++;
   release(B);
   release(A);
}
```

The best way to avoid deadlocks is to ensure that threads always acquire the locks in the same order. So if thread 2 acquired the locks in the order A and then B, it would stall while waiting for lock A without having first acquired lock B. This would enable thread 1 to acquire B and then eventually release both locks, allowing thread 2 to make progress. A livelock traps threads in an unending loop releasing and acquiring locks. Livelocks can be caused by code to back out of deadlocks.

the programmer has tried to implement a mechanism that avoids deadlocks. If the thread cannot obtain the second lock it requires, it releases the lock that it already holds. The two routines update1() and update2() each have an outer loop. Routine update1() acquires lock A and then attempts to acquire lock B, whereas update2() does this in the opposite order. This is a classic deadlock opportunity, and to avoid it, the developer has written some code that causes the held lock to be released if it is not possible to acquire the second lock. The routine canAquire(), in this example, returns immediately either having acquired the lock or having failed to acquire the lock.

Listing 4.14 **Two Threads in a Livelock**

Thread 1

```
void update1()
{
```

Thread 2

```
void update2()
{
```

```
int done=0;
while (!done)
{
  acquire(A);
  if ( canAcquire(B) )
  {
    variable1++;
    release(B);
    release(A);
    done=1;
  }
  else
  {
    release(A);
  }
}
```

```
int done=0;
while (!done)
{
  acquire(B);
  if ( canAcquire(A) )
  {
    variable2++;
    release(A);
    release(B);
    done=1;
  }
  else
  {
    release(B);
  }
}
```

Communication Between Threads and Processes

Condition Variables

Condition variables communicate readiness between threads by enabling a thread to be woken up when a condition becomes true. Without condition variables, the waiting thread would have to use some form of polling to check whether the condition had become true.

For example, the producer consumer model can be implemented using condition variables. Suppose an application has one producer thread and one consumer thread. The producer adds data onto a queue, and the consumer removes data from the queue. If there is no data on the queue, then the consumer needs to sleep until it is signaled that an item of data has been placed on the queue.

Producer Thread Adding an Item to the Queue

```
Acquire Mutex();
Add Item to Queue();
If ( Only One Item on Queue )
{
Signal Conditions Met();
}
Release Mutex();
```

The producer thread needs to signal a waiting consumer thread only if the queue was empty and it has just added a new item into that queue. If there were multiple items already on the queue, then the consumer thread must be busy processing those items and cannot be sleeping.

```
Acquire Mutex();
Repeat Item = 0;
If ( No Items on Queue() )
{
Wait on Condition Variable();
}
If (Item on Queue())
{
Item = remove from Queue();
}
```

```
Until ( Item != 0 );
Release Mutex();
```
The producer thread can use two types of wake-up calls: Either it can wake up a single thread or it can broadcast to all waiting threads.

Consumer Thread Code with Potential Lost Wake-Up Problem

```
Repeat
Item = 0;
If ( No Items on Queue() )
{
Acquire Mutex();
Wait on Condition Variable();
Release Mutex();
}
Acquire Mutex();
If ( Item on Queue() )
{
Item = remove from Queue();
 }
Release Mutex();
Until ( Item!=0 );
```

Signals and Events

Signals are a UNIX mechanism where one process can send a signal to another process and have a handler in the receiving process perform some task upon the receipt of the message. Many features of UNIX are implemented using signals. Stopping a running application by pressing ^C causes a SIGKILL signal to be sent to the process. Windows has a similar mechanism for events. The handling of keyboard presses and mouse moves are performed through the event mechanism. Pressing one of the buttons on the mouse will cause a click event to be sent to the target window. Signals and events are really optimized for sending limited or no data along with the signal, and as such they are probably not the best mechanism for communication when compared to other options

Installing and Using a Signal Handler

```
void signalHandler(void *signal)
{ ... }
int main()
{
installHandler( SIGNAL, signalHandler );
sendSignal( SIGNAL );
}
```

Message Queues

A message queue is a structure that can be shared between multiple processes. Messages can be placed into the queue and will be removed in the same order in which they were added. Constructing a message queue looks rather like constructing a shared memory segment. The first thing needed is a descriptor, typically the location of a file in the file system. This descriptor can either be used to create the message queue or be used to attach to an existing message queue. Once the queue is configured, processes can place messages into it or remove messages from it. Once the queue is finished, it needs to be deleted.

Creating and Placing Messages into a Queue ID = Open Message Queue Queue(Descriptor); Put Message in Queue(ID, Message); ... Close Message Queue(ID); Delete Message Queue(Description);
Using the descriptor for an existing message queue enables two processes to communicate by sending and receiving messages through the queue.
Opening a Queue and Receiving Messages ID=Open Message Queue ID(Descriptor); Message=Remove Message from Queue(ID); ... Close Message Queue(ID);

Named Pipes

UNIX uses pipes to pass data from one process to another. For example, the output from the command ls, which lists all the files in a directory, could be piped into the wc command, which counts the number of lines, words, and characters in the input. The combination of the two commands would be a count of the number of files in the directory. Named pipes provide a similar mechanism that can be controlled programmatically.
Setting Up and Writing into a Pipe Make Pipe(Descriptor); ID = Open Pipe(Descriptor); Write Pipe(ID, Message, sizeof(Message)); ... Close Pipe(ID); Delete Pipe(Descriptor);
Opening an Existing Pipe to Receive Messages ID=Open Pipe(Descriptor); Read Pipe(ID, buffer, sizeof(buffer)); ... Close Pipe(ID);

other approaches to Sharing Data Between Threads:

There are several other approaches to sharing data. For example, data can be written to a file to be read by another process at a later point. This might be acceptable if the data needs to be stored persistently or if the data will be used at some later point. Still, writing to disk presents a long latency operation, which is not the best mechanism if the purpose is purely communication

There is also operating system–specific approaches to sharing data between processes. Solaris doors allow one process to pass an item of data to another process and have the processed result returned. Doors are optimized for the round-trip and hence can be cheaper than using two different messages.

UNIT III SHARED MEMORY PROGRAMMING WITH OpenMP

OpenMP Execution Model – Memory Model – OpenMP Directives – Work-sharing Constructs - Library functions – Handling Data and Functional Parallelism – Handling Loops - Performance Considerations.

OpenMP

OpenMP is an API for shared-memory parallel programming. The "MP" in OpenMP stands for "multiprocessing,". OpenMP is designed for systems in which each thread or process can potentially have access to all available memory.

OpenMP Execution Model

OpenMP provides what's known as a "directives-based" shared-memory API. In C and C++, this means that there are special preprocessor instructions known as pragmas.

Pragmas are typically added to a system to allow behaviors that aren't part of the basic C specification. Compilers that don't support the pragmas are free to ignore them. This allows a program that uses the pragmas to run on platforms that don't support them.

Pragmas in C and C++ start with

#pragma

Example: Openmp 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", my_rank, thread_count);
22
23 }  /* Hello */
```

Compiling and running OpenMP programs

To compile this with gcc we need to include the -fopenmp option

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

To run the program, we specify the number of threads on the command line. For example, we might run the program with four threads and type

```
$  ./omp_hello  4
```

Output

Hello from thread 0 of 4

Hello from thread 1 of 4

Hello from thread 2 of 4

Hello from thread 3 of 4

The program

The OpenMP header file is omp.h. we specified the number of threads on the command line. In Line 9 we therefore use the strtol function from stdlib.h to get the number of threads.

Syntax: strtol

```
long strtol(
      const char*  number p     /* in  */,
      char**       end p        /* out */,
      int          base         /* in  */);
```

The first argument is a string—in our example, it's the command-line argument—and the last argument is the numeric base in which the string is represented—in our example, it's base 10. We won't make use of the second argument, so we'll just pass in a NULL pointer.

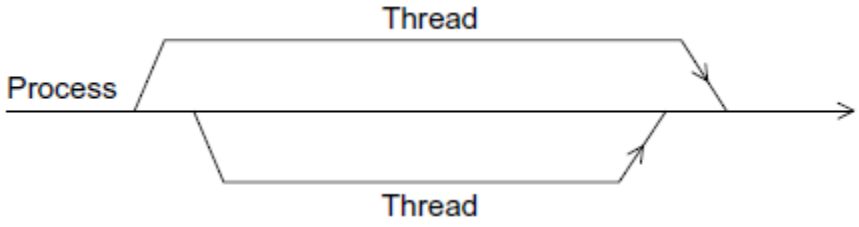
OpenMP pragmas always begin with

 # pragma omp

The first directive is a parallel directive, and, it specifies that the structured block of code that follows should be executed by multiple threads. Thread is short for thread of execution.

The name is meant to suggest a sequence of statements executed by a program. Threads are typically started or forked by a process, and they share most of the resources of the process that starts them, but each thread has its own stack and program counter.

When a thread completes execution it joins the process that started it. This terminology comes from diagrams that show threads as directed lines.



The parallel directive is simply

 # pragma omp parallel

and the number of threads that run the following structured block of code will be determined by the run-time system. If there are no other threads started, the system will typically run one thread on each available core.

Usually the number of threads is specified on the command line, so modify parallel directives with the num threads *clause.*

A **clause** in OpenMP is just some text that modifies a directive. The num threads clause can be added to a parallel directive. It allows the programmer to specify the number of threads that should execute the following block:

 # pragma omp parallel num_threads(thread count)

It should be noted that there may be system-defined limitations on the number of threads that a program can start.

Prior to the parallel directive, the program is using a single thread, the process started when the program started execution. When the program reaches the parallel directive, the original thread continues executing and thread count - 1 additional threads are started. In OpenMP, the collection of threads executing the parallel block—the original thread and the new threads—is called a **team**, the original thread is called the **master**, and the additional threads are called **slaves**.

Each thread in the team executes the block following the directive, so in the example, each thread calls the Hello function.

When the block of code is completed, when the threads return from the call to Hello—there's an **implicit barrier**. This means that a thread that has completed the block of code will wait for all the other threads in the team to complete the block—in the example, a thread that has completed the call to Hello will wait for all the other threads in the team to return.

When all the threads have completed the block, the slave threads will terminate and the master thread will continue executing the code that follows the block.

In the example, the master thread will execute the **return** statement in Line 14, and the program will terminate.

Since each thread has its own stack, a thread executing the Hello function will create its own private, local variables in the function.

In our example, when the function is called, each thread will get its rank or id and the number of threads in the team by calling the OpenMP functions omp_get_thread_num and omp_get_num_threads,respectively.

The rank or id of a thread is an **int** that is in the range 0, 1, … , thread count -1.

The syntax for these functions is

int omp_get_thread_num (**void**);

int omp_get_num_threads (**void**);

Since stdout is shared among the threads, each thread can execute the printf statement,printing its rank and the number of threads.

Error checking

Check strtol function call

In this example, we should definitely check for the presence of a command-line argument, and, if there is one, after the call to strtol we should check that the value is positive.

Check the number of threads created

We might also check that the number of threads actually created by the parallel directive is the same as thread count.

Compiler

If the compiler doesn't support OpenMP, it will just ignore the parallel directive. However, the attempt to include omp.h and the calls to omp_get_thread_ num and omp_get_num_threads will cause errors.

To handle these problems, we can check whether the preprocessor macro OPENMP is defined. If this is defined, we can include omp.h and make the calls to the OpenMP functions.

We might make the following modifications to our program.Instead of simply including omp.h in the line

#include <omp.h>

we can check for the definition of OPENMP before trying to include it:

```
#ifdef OPENMP
# include <omp.h>
#endif
```

Also, instead of just calling the OpenMP functions, we can first check whether OPENMP is defined:

```
# 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
```

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

MEMORY MODEL

OpenMP assumes that there is a place for storing and retrieving data that is available to all threads, called the memory. Each thread may have a temporary view of memory that it can use instead of memory to store data temporarily when it need not be seen by other threads.

Data can move between memory and a thread's temporary view, but can never move between temporary views directly, without going through memory. Each variable used within a parallel region is either shared or private. The variable names used within a parallel construct relate to the program variables visible at the point of the parallel directive, referred to as their "original variables". Each shared variable reference inside the construct refers to the original variable of the same name. For each private variable, a reference to the variable name inside the construct refers to a variable of the same type and size as the original variable, but private to the thread. That is, it is not accessible by other threads.

There are two aspects of memory system behavior relating to shared memory parallel programs: **coherence and consistency**.

Coherence refers to the behavior of the memory system when a single memory location is accessed by multiple threads.

Consistency refers to the ordering of accesses to different memory locations, observable from various threads in the system.

OpenMP doesn't specify any coherence behavior of the memory system. That is left to the underlying base language and computer system. OpenMP does not guarantee anything about the result of memory operations that constitute data races within a program.

A data race is defined to be accesses to a single variable by at least two threads, at least one of which is a write, not separated by a synchronization operation. OpenMP does guarantee certain consistency behavior, however. That behavior is based on the OpenMP flush operation.

The OpenMP flush operation is applied to a set of variables called the flush set. Memory operations for variables in the flush set that precede the flush in program execution order must be firmly lodged in memory and available to all threads before the flush completes, and memory operations for variables in the flush set, that follow a flush in program order cannot start until the flush completes.

A flush also causes any values of the flush set variables that were captured in the temporary view, to be discarded, so that later reads for those variables will come directly from memory. A flush without a list of variable names flushes all variables visible at that point in the program. A flush with a list flushes only the variables in the list. The OpenMP flush operation is the only way in an OpenMP program, to guarantee that a value will move between two threads.

In order to move a value from one thread to a second thread, OpenMP requires these four actions in exactly the following order:

1. the first thread writes the value to the shared variable,

2. the first thread flushes the variable.
3. the second thread flushes the variable and
4. the second thread reads the variable.

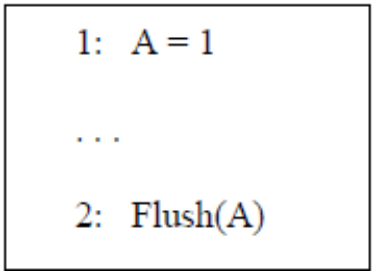


Figure: A write to shared variable A may complete as soon as point 1, and as late as point 2.

The flush operation and the temporary view allow OpenMP implementations to optimize reads and writes of shared variables. For example, consider the program fragment in Figure above. The write to variable A may complete as soon as point 1 in the figure.

However, the OpenMP implementation is allowed to execute the computation denoted as "…" in the figure, before the write to A completes. The write need not complete until point 2, when it must be firmly lodged in memory and available to all other threads. If an OpenMP implementation uses a temporary view, then a read of A during the "…" computation in Figure can be satisfied from the temporary view, instead of going all the way to memory for the value. So, flush and the temporary view together allow an implementation to hide both write and read latency.

A flush of all visible variables is implied 1) in a barrier region, 2) at entry and exit from parallel, critical and ordered regions, 3) at entry and exit from combined parallel work-sharing regions, and 4) during lock API routines.

The flushes associated with the lock routines were specifically added in the OpenMP specification, a distinct change to both 2.0 specifications, as discussed in the following section. A flush with a list is implied at entry to and exit from atomic regions, where the list contains the object being updated. The C and C++ languages include the volatile qualifier, which provides a consistency mechanism for C and C++ that is related to the OpenMP consistency mechanism.

When a variable is qualified with volatile, an OpenMP program must behave as if a flush operation with that variable as the flush set were inserted in the program. When a read is done for the variable, the program must behave as if a flush were inserted in the program at the sequence point prior to the read. When a write is done for the variable, the program must behave as if a flush were inserted in the program at the sequence point after the write.

Another aspect of the memory model is the accessibility of various memory locations. OpenMP has three types of accessibility: shared, private and threadprivate. Shared variables are accessible by all threads of a thread team and any of their descendant threads in nested parallel regions. Access to private variables is restricted. If a private variable X is created for one thread upon entry to a parallel region, the sibling threads in the same team, and their descendant threads, must not access it.

However, if the thread for which X was created encounters a new parallel directive (becoming the master thread for the inner team), it is permissible for the descendant threads in the inner team to access X, either directly as a shared variable, or through a pointer.

The difference between access by sibling threads and access by the descendant threads is that the variable X is guaranteed to be still available to descendant threads, while it might be popped off the stack before siblings can

access it. For a thread_private variable, only the thread to which it is private may access it, regardless of nested parallelism

OpenMP Directives

OpenMP directives exploit shared memory parallelism by defining various types of parallel regions. Parallel regions can include both iterative and non-iterative segments of program code.

The **#pragma omp** pragmas fall into these general categories:

1) The Parallel Directive

Defines a parallel region, which is code that will be executed by multiple threads in parallel. When a thread reaches a PARALLEL directive, it creates a team of threads and becomes the master of the team. The master is a member of that team and has thread number 0 within that team.

Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code. There is an implied barrier at the end of a parallel region. Only the master thread continues execution past this point. If any thread terminates within a parallel region, all threads in the team will terminate, and the work done up until that point is undefined.

The number of threads in a parallel region is determined by the following factors, in order of precedence:

- Evaluation of the **IF** clause
- Setting of the **NUM_THREADS** clause
- Use of the **omp_set_num_threads()** library function
- Setting of the **OMP_NUM_THREADS** environment variable
- Implementation **default** - usually the number of CPUs on a node, though it could be dynamic

Threads are numbered from 0 (master thread) to N-1.

Syntax

```
#pragma omp parallel [clauses]
{
        code_block
}
```

where,

clause (optional). Zero or more clauses.

Nested Parallel Regions:

- Use the **omp_get_nested()** library function to determine if nested parallel regions are enabled.
- The two methods available for enabling nested parallel regions (if supported) are:
  1. The **omp_set_nested()** library routine
  2. Setting of the **OMP_NESTED** environment variable to TRUE
- If not supported, a parallel region nested within another parallel region results in the creation of a new team, consisting of one thread, by default.

Clauses:

- **IF clause:** If present, it must evaluate to .TRUE. (Fortran) or non-zero (C/C++) in order for a team of threads to be created. Otherwise, the region is executed serially by the master thread.

Restrictions:

- A parallel region must be a structured block that does not span multiple routines or code files
- It is illegal to branch (goto) into or out of a parallel region

- Only a single IF clause is permitted
- Only a single NUM_THREADS clause is permitted
- A program must not depend upon the ordering of the clauses

Example: Program

```
#include <stdio.h>
#include <omp.h>
int main() {
  #pragma omp parallel num_threads(4)
  {
    int i = omp_get_thread_num();
    printf_s("Hello from thread %d\n", i);
  }
}
```

Output

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

2. The parallel for Directive

- Like the parallel directive, the parallel for directive forks a team of threads to execute the following structured block.
- The structured block following the parallel for directive must be a for loop.

With the parallel for directive the system parallelizes the for loop by dividing the iterations of the loop among the threads. The parallel for directive is therefore very different from the parallel directive, because in a block that is preceded by a parallel directive, in general, the work must be divided among the threads by the threads themselves.

- If there are *m* iterations, then roughly the first *m*/thread count are assigned to thread 0, the next *m*/thread count are assigned to thread 1, and so on.

Syntax

#pragma omp parallel for [clauses]

for_statement

where,

clause (optional),Zero or more clauses.

Example code

```
int x,y;
#pragma omp parallel for
for(x=0; x < width; x++)
{
        for(y=0; y < height; y++)
        {
                finalImage[x][y] = RenderPixel(x,y, &sceneData);
        }
}
```

Caveats

OpenMP will only parallelize for loops. It won't parallelize while loops or do-while loops. This may not seem to be too much of a limitation, since any code that uses a while loop or a do-while loop can be converted to equivalent code that uses a for loop instead.

OpenMP will only parallelize for loops for which the number of iterations can be determined.

For example, the "**infinite loop**"

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

cannot be parallelized.

Similarly, the loop

```
for (i = 0; i < n; i++) {
        if ( . . . ) break;
        . . .
}
```

cannot be parallelized, since the number of iterations can't be determined from the for statement alone.

OpenMP will only parallelize for loops that are in canonical form. Loops in canonical form take one of the forms shown below:

```
                                    index++
                                    ++index
                    index < end     index--
                    index <= end    --index
for | index = start ; index >= end ; index += incr |
                    index > end     index -= incr
                                    index = index + incr
                                    index = incr + index
                                    index = index - incr
```

The variable index must have integer or pointer type (e.g., it can't be a **float**). The expressions start, end, and incr must have a compatible type. For example,

if index is a pointer, then incr must have integer type. The expressions start, end, and incr must not change during execution of the loop. During execution of the loop, the variable index can only be modified by the "increment expression" in the **for** statement.

Data dependences

1. OpenMP compilers don't check for dependences among iterations in a loop that's being parallelized with a parallel **for** directive. It's up to us, the programmers, to identify these dependences.

2. A loop in which the results of one or more iterations depend on other iterations *cannot,* in general, be correctly parallelized by OpenMP.

Finding loop-carried dependences

For example, in the loop

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

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

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

since the computation of x[i] and its subsequent use will always be assigned to the same thread. Also observe that at least one of the statements must write or update the variable in order for the statements to represent a dependence, so in order to detect a loopcarried dependence, we should only concern ourselves with variables that are updatedby the loop body. That is, we should look for variables that are read or written in one iteration, and written in another.

3) The atomic directive

The atomic directive ensures that a specific memory location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads.

The atomic directive supports no OpenMP clauses.

Syntax

```
#pragma omp atomic
        expression
```

Example

```
#include <stdio.h>
#include <omp.h>
#define MAX 10

int main() {
  int count = 0;
  #pragma omp parallel num_threads(MAX)
  {
    #pragma omp atomic
    count++;
  }
```

```
  printf_s("Number of threads: %d\n", count);
}
```

Output

Number of threads: 10

4) Critical Directive

Specifies that code is only executed on one thread at a time. OpenMP *does* provide the option of adding a name to a critical directive:

Syntax

 # pragma omp critical(name)

Two blocks protected with critical directives with different names *can* be executed simultaneously. The names are set during compilation. When we want to allow simultaneous access to the same block of code by different threads, the named critical directive isn't sufficient.The alternative is to use locks.

The use of a lock can be roughly described by the following pseudocode:

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

Destroy the lock data structure;

OpenMP has two types of locks:

- **simple locks and**
- **nested locks.**

A simple lock can only be set once before it is unset, while a nested lock can be set multiple times by the same thread before it is unset.

Example

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define SIZE 10

int main()
{
   int i;
   int max;
   int a[SIZE];
    for (i = 0; i < SIZE; i++)
   {
      a[i] = rand();
      printf_s("%d\n", a[i]);
   }

   max = a[0];
#pragma omp parallel for num_threads(4)
      for (i = 1; i < SIZE; i++)
```

```
    {
      if (a[i] > max)
      {
        #pragma omp critical
        {
          // compare a[i] and max again because max
          // could have been changed by another thread after
          // the comparison outside the critical section
          if (a[i] > max)
            max = a[i];
        }
      }
    }

  printf_s("max = %d\n", max);
}
```

Output

```
41
18467
6334
26500
19169
15724
11478
29358
26962
24464
```

max = 29358

5) Barrier Directive

A barrier directive will cause the threads in a team to block until all the threads have reached the directive.

Syntax

#pragma omp barrier

6) Master Directive

Specifies that only the master thread should execute a section of the program.

Syntax

 # pragma omp master

Example

```
#include <omp.h>
#include <stdio.h>
 int main( )
{
   int a[5], i;

   #pragma omp parallel
   {
      // Perform some computation.
      #pragma omp for
      for (i = 0; i < 5; i++)
         a[i] = i * i;

      // Print intermediate results.
      #pragma omp master
```

```
        for (i = 0; i < 5; i++)
            printf_s("a[%d] = %d\n", i, a[i]);
      // Wait.
    #pragma omp barrier
      // Continue with the computation.
    #pragma omp for
    for (i = 0; i < 5; i++)
        a[i] += i;
  }
}
```

Output

a[0] = 0

a[1] = 1

a[2] = 4

a[3] = 9

a[4] = 16

7) Single Directive

Lets you specify that a section of code should be executed on a single thread, not necessarily the master thread.

Syntax

```
#pragma omp single [clauses]
{
  code_block
}
```

Where,clause (optional),Zero or more clauses.

The **single** directive supports the following OpenMP clauses:

- copyprivate
- firstprivate
- nowait
- private

The master directive lets you specify that a section of code should be executed only on the master thread.

With the single directive

 # pragma omp single

Execute action;

Next action;

The run-time system will choose a single thread to execute the action. The other threads will wait before proceeding to Next action.

With the master directive

 # pragma omp master

Execute action;

Next action;

the master thread (thread 0) will execute the action.

Example

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

int main() {
  #pragma omp parallel num_threads(2)
  {
    #pragma omp single
    // Only a single thread can read the input.
    printf_s("read input\n");
// Multiple threads in the team compute the results.
    printf_s("compute results\n");

    #pragma omp single
    // Only a single thread can write the output.
    printf_s("write output\n");
  }
}
```

Output

```
read input
compute results
compute results
write output
```

WORK-SHARING CONSTRUCTS

Work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it. A work-sharing construct must be enclosed within a parallel region in order for the directive to execute in parallel. When a work-sharing construct is not enclosed within a parallel region, it is treated as though one thread executes it. The work-sharing directives do not launch new threads, and there is no implied barrier on entry to a work-sharing construct.Used to specify how to assign independent work to one or all of the threads.

- ***omp for* or *omp do***: used to split up loop iterations among the threads, also called loop constructs.
- ***sections*:** assigning consecutive but independent code blocks to different threads
- ***single*:** specifying a code block that is executed by only one thread, a barrier is implied in the end
- ***master*:** similar to single, but the code block will be executed by the master thread only and no barrier implied in the end.

Example:

Initialize the value of a large array in parallel, using each thread to do part of the work

```
int main(int argc, char **argv)
{
   int a[100000];

   #pragma omp parallel for
   for (int i = 0; i < 100000; i++) {
      a[i] = 2 * i;
   }
   return 0;
```

```
}
```

The loop counter i is declared inside the parallel for loop, which gives each thread a unique and private version of the variable.

Clauses

Since OpenMP is a shared memory programming model, most variables in OpenMP code are visible to all threads by default. But sometimes private variables are necessary to avoid race conditions and there is a need to pass values between the sequential part and the parallel region (the code block executed in parallel), so data environment management is introduced as *data sharing attribute clauses* by appending them to the OpenMP directive.

The different types of clauses are:

Data sharing attribute clauses

- ***shared*:** The data within a parallel region is shared, which means visible and accessible by all threads simultaneously. By default, all variables in the work sharing region are shared except the loop iteration counter.
- ***private*:** The data within a parallel region is private to each thread, which means each thread will have a local copy and use it as a temporary variable. A private variable is not initialized and the value is not maintained for use outside the parallel region. By default, the loop iteration counters in the OpenMP loop constructs are private.
- ***default*:** Allows the programmer to state that the default data scoping within a parallel region will be either *shared*, or *none* for C/C++, or *shared*, *firstprivate*, *private*, or *none* for Fortran. The *none* option forces the programmer to declare each variable in the parallel region using the data sharing attribute clauses.
- ***firstprivate*:** like *private* except initialized to original value.

- ***lastprivate*:** like *private* except original value is updated after construct.
- ***reduction*:** a safe way of joining work from all threads after construct.

Synchronization clauses

- ***critical*:** The enclosed code block will be executed by only one thread at a time, and not simultaneously executed by multiple threads. It is often used to protect shared data from race conditions.
- ***atomic*:** The memory update (write, or read-modify-write) in the next instruction will be performed atomically. It does not make the entire statement atomic; only the memory update is atomic. A compiler might use special hardware instructions for better performance than when using *critical*.
- ***ordered*:** The structured block is executed in the order in which iterations would be executed in a sequential loop
- ***barrier*:** Each thread waits until all of the other threads of a team have reached this point. A work-sharing construct has an implicit barrier synchronization at the end.
- ***nowait*:** Specifies that threads completing assigned work can proceed without waiting for all threads in the team to finish. In the absence of this clause, threads encounter a barrier synchronization at the end of the work sharing construct.

Scheduling clauses

- ***schedule(type, chunk)*:** This is useful if the work sharing construct is a do-loop or for-loop. The iteration(s) in the work sharing construct are assigned to threads according to the scheduling method defined by this clause.
- The three types of scheduling are:

1. ***static*:** Here, all the threads are allocated iterations before they execute the loop iterations. The iterations are divided among threads equally by default. However, specifying an integer for the parameter *chunk* will allocate chunk number of contiguous iterations to a particular thread.
2. ***dynamic*:** Here, some of the iterations are allocated to a smaller number of threads. Once a particular thread finishes its allocated iteration, it returns to get another one from the iterations that are left. The parameter *chunk* defines the number of contiguous iterations that are allocated to a thread at a time.
3. ***guided*:** A large chunk of contiguous iterations are allocated to each thread dynamically (as above). The chunk size decreases exponentially with each successive allocation to a minimum size specified in the parameter *chunk*

IF control

- ***if*:** This will cause the threads to parallelize the task only if a condition is met. Otherwise the code block executes serially.

Initialization

- ***firstprivate*:** the data is private to each thread, but initialized using the value of the variable using the same name from the master thread.
- ***lastprivate*:** the data is private to each thread. The value of this private data will be copied to a global variable using the same name outside the parallel region if current iteration is the last iteration in the parallelized loop. A variable can be both *firstprivate* and *lastprivate*.
- ***threadprivate*:** The data is a global data, but it is private in each parallel region during the runtime. The difference between *threadprivate* and *private* is the global scope associated with threadprivate and the preserved value across parallel regions.

Data copying

- ***copyin*:** similar to *firstprivate* for *private* variables, *threadprivate* variables are not initialized, unless using *copyin* to pass the value from the corresponding global variables. No *copyout* is needed because the value of a threadprivate variable is maintained throughout the execution of the whole program.
- ***copyprivate*:** used with *single* to support the copying of data values from private objects on one thread (the *single* thread) to the corresponding objects on other threads in the team.

Reduction

- ***reduction(operator | intrinsic : list)*:** The variable has a local copy in each thread, but the values of the local copies will be summarized (reduced) into a global shared variable. This is very useful if a particular operation (specified in *operator* for this particular clause) on a variable runs iteratively, so that its value at a particular iteration depends on its value at a prior iteration. The steps that lead up to the operational increment are parallelized, but the threads updates the global variable in a thread safe manner.

Others

- ***flush*:** The value of this variable is restored from the register to the memory for using this value outside of a parallel part
- ***master*:** Executed only by the master thread. No implicit barrier; other team members (threads) not required to reach.

User-level runtime routines

Used to modify/check the number of threads, detect if the execution context is in a parallel region, how many processors in current system, set/unset locks, timing functions, etc.

Environment variables

A method to alter the execution features of OpenMP applications. Used to control loop iterations scheduling, default number of threads, etc. For

example, *OMP_NUM_THREADS* is used to specify number of threads for an application.

LIBRARY FUNCTIONS

OpenMP provides several runtime library routines to assist you in managing your program in parallel mode. Many of these runtime library routines have corresponding environment variables that can be set as defaults. The runtime library routines enable you to dynamically change these factors to assist in controlling your program. In all cases, a call to a runtime library routine overrides any corresponding environment variable.

omp_set_num_threads(*num_threads*)

- Sets the number of threads to use for subsequent parallel regions.

omp_get_num_threads()

- Returns the number of threads that are being used in the current parallel region.

omp_get_max_threads()

- Returns the maximum number of threads that are available for parallel execution

omp_get_thread_num()

- Determines the unique thread number of the thread currently executing this section of code.

omp_get_num_procs()

- Determines the number of processors available to the program.

omp_in_parallel()

- Returns .true. if called within the parallel region executing in parallel; otherwise returns .false.

omp_get_dynamic()

- Returns .true. if dynamic thread adjustment is enabled, otherwise returns .false..

omp_get_nested()

- Returns .true. if nested parallelism is enabled, otherwise returns .false.

omp_set_nested(*nested*)

- Enables or disables nested parallelism. If *nested* is .true., nested parallelism is enabled. If *nested* is .false., nested parallelism is disabled. Nested parallelism is disabled by default.

omp_init_lock(*lock*)

- Initializes the lock associated with *lock* for use in subsequent calls

omp_destroy_lock(*lock*)

- Causes the lock associated with *lock* to become undefined.

omp_set_lock(*lock*)

- Forces the executing thread to wait until the lock associated with *lock* is available.
- The thread is granted ownership of the lock when it becomes available.

omp_unset_lock(*lock*)

- Releases the executing thread from ownership of the lock associated with *lock*.

omp_test_lock(*lock*)

- Attempts to set the lock associated with *lock*. If successful, returns .true., otherwise returns .false..

mp_init_nest_lock(lock)

- Initializes the nested lock associated with *lock* for use in the subsequent calls.

omp_destroy_nest_lock(*lock*)

- Causes the nested lock associated with *lock* to become undefined.

omp_set_nest_lock(lock)

- Forces the executing thread to wait until the nested lock associated with *lock*is available. The thread is granted ownership of the nested lock when it becomes available.

omp_unset_nest_lock(lock)

- Releases the executing thread from ownership of the nested lock associated with *lock* if the nesting count is zero. Behavior is undefined if the executing thread does not own the nested lock associated with *lock*

omp_test_nest_lock(*lock*)

- Attempts to set the nested lock associated with *lock*. If successful, returns the nesting count, otherwise returns zero.

omp_get_wtime()

- Returns a double-precision value equal to the elapsed wallclock time (in seconds) relative to an arbitrary reference time. The reference time does not change during program execution.

omp_get_wtick()

- Returns a double-precision value equal to the number of seconds between successive clock ticks.

HANDLING DATA AND FUNCTIONAL PARALLELISM

Data Parallelism

Data parallelism is a form of parallelization across multiple processors in parallel computing environments. It focuses on distributing the data across different nodes, which operate on the data in parallel. It can be applied on regular data structures like arrays and matrices by working on each element in parallel. It contrasts to task parallelism as another form of parallelism.

A data parallel job on an array of 'n' elements can be divided equally among all the processors. Let us assume we want to sum all the elements of the given array and the time for a single addition operation is Ta time units.

In the case of sequential execution, the time taken by the process will be n*Ta time units as it sums up all the elements of an array. On the other hand, if we execute this job as a data parallel job on 4 processors the time taken would reduce to (n/4)*Ta + Merging overhead time units. Parallel execution results in a speedup of 4 over sequential execution. One important thing to note is that the locality of data references plays an important part in evaluating the performance of a data parallel programming model. Locality of data depends on the memory accesses performed by the program as well as the size of the cache.

In a multiprocessor system executing a single set of instructions (SIMD), data parallelism is achieved when each processor performs the same task on different pieces of distributed data. In some situations, a single execution thread controls operations on all pieces of data. In others, different threads control the operation, but they execute the same code.

For instance, consider matrix multiplication and addition in a sequential manner as discussed in the example.

Example

Below is the sequential pseudo-code for multiplication and addition of two matrices where the result is stored in the matrix C. The pseudo-code for multiplication calculates the dot product of two matrices A, B and stores the result into the output matrix C.

If the following programs were executed sequentially, the time taken to calculate the result would be of the $O(n^3)$ and $O(n)$ for multiplication and addition respectively.

//Matrix Multiplication

```
for(i=0; i<row_length_A; i++)
{
    for (k=0; k<column_length_B; k++)
    {
        sum = 0;
        for (j=0; j<column_length_A; j++)
        {
            sum += A[i][j]*B[j][k];
        }
        C[i][k]=sum;
    }
}
```

//Array addition

```
for(i=0;i<n;i++) {
  c[i]=a[i]+b[i];
  }
```

We can exploit data parallelism in the preceding codes to execute it faster as the arithmetic is loop independent. Parallelization of the matrix multiplication code is achieved by using OpenMP.

An OpenMP directive, "omp parallel for" instructs the compiler to execute the code in the for loop in parallel. For multiplication, we can divide matrix A and B into blocks along rows and columns respectively. This allows us to calculate every element in matrix C individually thereby making the task parallel.

/Matrix multiplication in parallel

```
#pragma omp parallel for schedule(dynamic,1) collapse(2)
for(i=0; i<row_length_A; i++){
      for (k=0; k<column_length_B; k++){
            sum = 0;
            for (j=0; j<column_length_A; j++){
                  sum += A[i][j]*B[j][k];
            }
            C[i][k]=sum;
      }
}
```

It can be observed from the example that a lot of processors will be required as the matrix sizes keep on increasing. Keeping the execution time low is the priority but as the matrix size increases, we are faced with other constraints like complexity of such a system and its associated costs. Therefore, constraining the number of processors in the system, we can still apply the same principle and divide the data into bigger chunks to calculate the product of two matrices.

For addition of arrays in a data parallel implementation, let's assume a more modest system with two Central Processing Units (CPU) A and B, CPU A could add all elements from the top half of the arrays, while CPU B could add all elements from the bottom half of the arrays. Since the two processors work in parallel, the job of performing array addition would take one half the time of performing the same operation in serial using one CPU alone.

The program expressed in pseudocode below—which applies some arbitrary operation, `foo`, on every element in the array `d`—illustrates data parallelism:

```
if CPU = "a"
```

```
    lower_limit := 1
    upper_limit := round(d.length/2)
else if CPU = "b"
    lower_limit := round(d.length/2) + 1
    upper_limit := d.length

for i from lower_limit to upper_limit by 1
    foo(d[i])
```

Steps to parallelization

The process of parallelizing a sequential program can be broken down into four discrete steps.

Type	Description
Decomposition	The program is broken down into tasks, the smallest exploitable unit of concurrence.
Assignment	Tasks are assigned to processes.
Orchestration	Data access, communication, and synchronization of processes.
Mapping	Processes are bound to processors.

Functional Parallelism

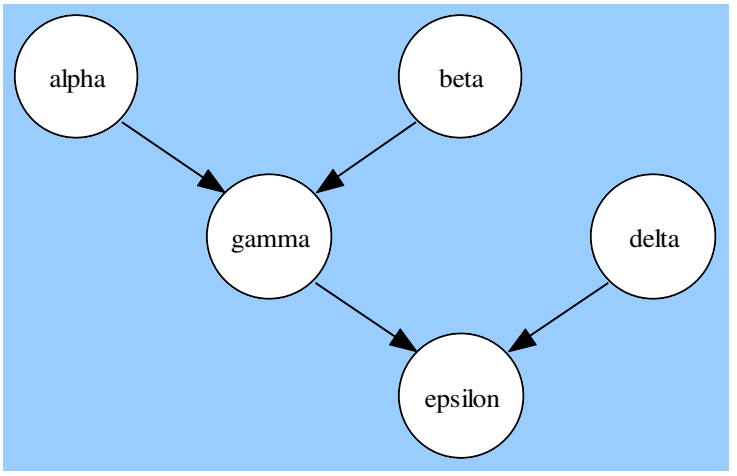
- OpenMP allows us to assign different threads to different portions of code (functional parallelism)

Functional Parallelism Example

```
v = alpha();
  w = beta();
  x = gamma(v, w);
```

```
y = delta();
printf ("%6.2f\n", epsilon(x,y));
```

May execute alpha,beta, and delta in parallel



parallel sections Pragma

- Precedes a block of *k* blocks of code that may be executed concurrently by *k* threads

Syntax:

#pragma omp parallel sections

section Pragma

- Precedes each block of code within the encompassing block preceded by the parallel sections pragma
- May be omitted for first parallel section after the parallel sections pragma

Syntax:

#pragma omp section

Example of parallel sections

#pragma omp parallel sections

```
  {
#pragma omp section  /* Optional */
    v = alpha();
#pragma omp section
    w = beta();
#pragma omp section
    y = delta();
  }
  x = gamma(v, w);
  printf ("%6.2f\n", epsilon(x,y));
```

Another Approach

Execute alpha and beta in parallel.

Execute gamma and delta in parallel.

sections Pragma

- Appears inside a parallel block of code
- Has same meaning as the **parallel sections** pragma
- If multiple **sections** pragmas inside one parallel block, may reduce fork/join costs

Use of sections Pragma

```
#pragma omp parallel
  {
  #pragma omp sections
    {
      v = alpha();
    #pragma omp section
      w = beta();
    }
```

```
  #pragma omp sections
    {
      x = gamma(v, w);
    #pragma omp section
      y = delta();
    }
  }
  printf ("%6.2f\n", epsilon(x,y));
```

HANDLING LOOPS

If there are n iterations in the serial loop, then in the parallel loop the first n/thread count are assigned to thread 0, the next n/thread count are assigned to thread 1, and so on.

For example, suppose we want to parallelize the loop

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

the time required by the call to *f* is proportional to the size of the argument i. Then a block partitioning of the iterations will assign much more work to thread thread count-1 than it will assign to thread 0.

A better assignment of work to threads might be obtained with a cyclic partitioning of the iterations among the threads. In a cyclic partitioning, the iterations are assigned, one at a time, in a “round-robin” fashion to the threads. Suppose *t* D thread count. Then a cyclic partitioning will assign the iterations as follows:

Thread	Iterations
0	0, n/t, 2n/t, ...
1	1, n/t+1, 2n/t+1, ...
⋮	⋮
t−1	t−1, n/t+t−1, 2n/t+t−1, ...

When we ran the program with *n* D 10,000 and one thread, the run-time was 3.67 seconds. When we ran the program with two threads and the default assignment—iterations 0–5000 on thread 0 and iterations 5001–10,000 on thread 1—the run-time was 2.76 seconds. This is a speedup of only 1.33. However, when we ran the program with two threads and a cyclic assignment, the run-time was decreased to 1.84 seconds.

This is a speedup of 1.99 over the one-thread run and a speedup of 1.5 over the two-thread block partition!

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

The schedule clause

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

```
sum = 0.0;
# pragma omp parallel for num threads(thread count)
reduction(+:sum)
for (i = 0; i <= n; i++)
sum += f(i);
```

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

```
sum = 0.0;
# pragma omp parallel for num threads(thread count)
        reduction(+:sum) schedule(static,1)
for (i = 0; i <= n; i++)
sum += f(i)
```

The schedule clause has the form

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

The type can be any one of the following:

Static:

The iterations can be assigned to the threads before the loop is executed.

dynamic or guided

The iterations are assigned to the threads while the loop is executing, so after a thread completes its current set of iterations, it can request more from the run-time system.

auto

The compiler and/or the run-time system determine the schedule.

runtime

The schedule is determined at run-time.

- The chunksize is a positive integer. In OpenMP, a **chunk of iterations** is a block of iterations that would be executed consecutively in the serial loop.
- The number of iterations in the block is the chunksize. Only static, dynamic, and guided schedules can have a chunksize..

The static schedule type

For a static schedule, the system assigns chunks of (chunksize) iterations to each thread in a round-robin fashion.

As an example, suppose we have 12 iterations, 0, 1, . . .,11, and three threads. Then if schedule(**static,1**) is used in the parallel for or for directive, the iterations will be assigned as

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

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

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

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

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

The chunksize can be omitted. If it is omitted, the chunksize is approximately total iterations/thread count.

Dynamic schedule

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

Guided schedule

In a guided schedule, each thread also executes a chunk, and when a thread finishes a chunk, it requests another one. However, in a guided schedule, as chunks are completed, the size of the new chunks decreases.

For example, on one of our systems, if we run the trapezoidal rule program with the parallel **for** directive and a schedule(guided) clause, then when *n* D 10,000 and thread count D 2, the iterations are assigned as :

The first chunk has size 9999=2 ~ 5000, since there are 9999 unassigned iterations.The second chunk has size 4999=2 ~ 2500, and so on.

In a guided schedule, if no chunksize is specified, the size of the chunks decreases down to 1. If chunksize is specified, it decreases down to chunksize,with the exception that the very last chunk can be smaller than chunksize.

The runtime schedule type

Environment Variables

Environment variables are named values that can be accessed by a running program. That is, they're available in the program's *environment.* Some commonly used environment variables are:

PATH

HOME and

SHELL.

PATH:

The PATH variable specifies which directories the shell should search when it's looking for an executable. It's usually defined in both Unix and Windows.

HOME:

The HOME variable specifies the location of the user's home directory, and the SHELL variable specifies the location of the executable for the user's shell. These

are usually defined in Unix systems. In both Unix-like systems (e.g., Linux and Mac OS X) and Windows, environment variables can be examined and specified on the command line. In Unix-like systems, you can use the shell's command line. As an example, if we're using the bash shell, we can examine the value of an environment variable by typing

$ echo $PATH

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

$ export TEST VAR="hello"

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

$ export OMP SCHEDULE="static,1"

Now, when we start executing our program, the system will schedule the iterations of the **for** loop as if we had the clause schedule(**static**,1) modifiying the parallel **for** directive.

Selecting a schedule

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

- If each iteration of the loop requires roughly the same amount of computation, then it's likely that the default distribution will give the best performance.

- If the cost of the iterations decreases (or increases) linearly as the loop executes,then a static schedule with small chunksizes will probably give the best performance.
- If the cost of each iteration can't be determined in advance, then it may make sense to explore a variety of scheduling options. The schedule(runtime) clause can be used here, and the different options can be explored by running the program with different assignments to the environment variable OMP SCHEDULE.

PERFORMANCE CONSIDERATIONS

Once you have a correct, working OpenMP program, it is worth considering its overall performance.

There are some general techniques that you can utilize to improve the efficiency and scalability of an OpenMP application, as well as techniques specific to the Sun platforms.

The following are some general techniques for improving performance of OpenMP applications.

Minimize synchronization.

- Avoid or minimize the use of **BARRIER**, **CRITICAL** sections, **ORDERED** regions, and locks.
- Use the **NOWAIT** clause where possible to eliminate redundant or unnecessary barriers. For example, there is always an implied barrier at the end of a parallel region. Adding **NOWAIT** to a final **DO** in the region eliminates one redundant barrier.
- Use named **CRITICAL** sections for fine-grained locking.

- Use explicit **FLUSH** with care. Flushes can cause data cache restores to memory, and subsequent data accesses may require reloads from memory, all of which decrease efficiency.

By default, idle threads will be put to sleep after a certain time out period. It could be that the default time out period is not sufficient for your application, causing the threads to go to sleep too soon or too late. The **SUNW_MP_THR_IDLE** environment variable can be used to override the default time out period, even up to the point where the idle threads will never be put to sleep and remain active all the time.

Parallelize at the highest level possible, such as outer **DO/FOR** loops. Enclose multiple loops in one parallel region. In general, make parallel regions as large as possible to reduce parallelization overhead.

For example:

This construct is less efficient:

```
!$OMP PARALLEL
 ....
 !$OMP DO
   ....
  !$OMP END DO
  ....
!$OMP END PARALLEL
!$OMP PARALLEL
  ....
   !$OMP DO
     ....
   !$OMP END DO
  ....
```

```
!$OMP END PARALLEL
```

Efficient one:

```
!$OMP PARALLEL
  ....
  !$OMP DO
    ....
  !$OMP END DO

  .....

  !$OMP DO
    ....
  !$OMP END DO
!$OMP END PARALLEL
```

Use **PARALLEL DO/FOR** instead of worksharing **DO/FOR** directives in parallel regions. The **PARALLEL DO/FOR** is implemented more efficiently than a general parallel region containing possibly several loops.

For example:

*This construct is **less efficient:***

```
!$OMP PARALLEL
  !$OMP DO
    .....
  !$OMP END DO
!$OMP END PARALLEL
```

than this one:

```
!$OMP PARALLEL DO
  ....
!$OMP END PARALLEL
```

- Use **MASTER** instead of **SINGLE** wherever possible.
- The **MASTER** directive is implemented as an **IF**-statement with no implicit **BARRIER** : **IF(omp_get_thread_num() == 0) {...}**
- The **SINGLE** directive is implemented similar to other worksharing constructs. Keeping track of which thread reached **SINGLE** first adds additional runtime overhead. There is an implicit **BARRIER** if **NOWAIT** is not specified. It is less efficient.

Choose the appropriate loop scheduling.

- **STATIC** causes no synchronization overhead and can maintain data locality when data fits in cache. However, **STATIC** may lead to load imbalance.
- **DYNAMIC,GUIDED** incurs a synchronization overhead to keep track of which chunks have been assigned. And, while these schedules could lead to poor data locality, they can improve load balancing. Experiment with different chunk sizes.

Use LASTPRIVATE with care, as it has the potential of high overhead.

- Data needs to be copied from private to shared storage upon return from the parallel construct.
- The compiled code checks which thread executes the logically last iteration. This imposes extra work at the end of each chunk in a parallel **DO/FOR**. The overhead adds up if there are many chunks.

Use efficient thread-safe memory management.

- Applications could be using **malloc()** and **free()** explicitly, or implicitly in the compiler-generated code for dynamic/allocatable arrays, vectorized intrinsics, and so on.
- The thread-safe **malloc()** and **free()** in **libc** have a high synchronization overhead caused by internal locking. Faster versions can be found in the **libmtmalloc** library. Link with **-lmtmalloc** to use **libmtmalloc**.

Small data cases may cause OpenMP parallel loops to underperform. Use the **IF** clause on **PARALLEL** constructs to indicate that a loop should run parallel only in those cases where some performance gain can be expected.

When possible, merge loops

```
!$omp parallel do
 do i = ...
```

statements_1

```
 end do
!$omp parallel do
 do i = ...
```

statements_2

```
 end do
```

Merge the above into a single loop

```
!$omp parallel do
  do i = ...
```

statements_1

statements_2

```
  end do
```

Try nested parallelism if your application lacks scalability beyond a certain level.

UNIT IV DISTRIBUTED MEMORY PROGRAMMING WITH MPI

MPI program execution – MPI constructs – libraries – MPI send and receive – Point-to-point and Collective communication – MPI derived datatypes – Performance evaluation

MPI PROGRAM EXECUTION

In message-passing programs, a program running on one core-memorypair is usually called a **process**, and two processes can communicate by calling functions:

- one process calls a send function and
- the other calls a receive function

Theimplementation of message-passing using here is called **MPI**, which is anabbreviation of **Message-Passing Interface**.

MPIdefines a library of functions that can be called from C, C++, and Fortranprograms. There are some "global" communication functions that can involve morethan two processes. These functions are called **collective communications.**

In parallel programming, it's common for the processesto be identified by nonnegative integer ranks. So if there are p processes, the processeswill have ranks 0, 1, 2, : : : , p-1.

Compilation and execution

Many systems use a command called mpicc for compilation

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

mpicc is a script that's a **wrapper** for the C compiler.

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

Many systems also support program startup with mpiexec:

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

So to run the program with one process,

$ mpiexec-n 1 ./mpi_ hello

and to run the program with four processes,

$ mpiexec-n 4 ./mpi_hello

```
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", my_rank,
           comm_sz);
23       for (int q = 1; q < comm_sz; q++) {
24          MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
25             0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26          printf("%s\n", greeting);
27       }
28    }
29
30    MPI_Finalize();
31    return 0;
32 }  /* main */
```

Output

With one process the program's output would be

Greetings from process 0 of 1!

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!

MPI programs

This *is* a C program.It includes the standard C header files stdio.h and string.h. It also has a main function just like any other C program.

Line 3 includes the mpi.h header file. This contains prototypes of MPI functions, macro definitions, type definitions, and so on;it contains all the definitions and declarations needed for compiling an MPI program.

All of the identifiers defined by MPI start with the string MPI . The first letter following the underscore is capitalized for function names and MPI-defined types. All of the letters in MPI-defined macros and constants are capitalized.

MPI_ Init

In Line 12 the call to MPI_Init tells the MPI system to do all of the necessary setup.

For example, it might allocate storage for message buffers, and it might decide which process gets which rank. No other MPI functions should be called before the program calls MPI Init.

Syntax

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

The arguments, argc_p and argv_p, are pointers to the arguments to main, argc, and argv. However, when the program doesn't use these arguments, just pass NULL for both.

MPI_Finalize

In Line 30 the call to MPI Finalize tells the MPI system that we're done using MPI, and that any resources allocated for MPI can be freed.

Syntax

int MPI_Finalize(void);

No MPI functions should be called after the call to MPI Finalize.

Basic outline of a MPI program:

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

Communicators, MPI_Comm_size and MPI_Comm_rank

In MPI a **communicator** is a collection of processes that can send messages to each other. One of the purposes of MPI_Init is to define a communicator that consists of all of the processes started by the user when she started the program.

This communicator is called MPI COMM WORLD. The function calls in Lines 13 and 14 are getting information about MPI COMM WORLD.

Syntax

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

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

For both functions, the first argument is a communicator and has the special type defined by MPI for communicators, MPI Comm.

MPI_Comm_size returns in its second argument the number of processes in the communicator.

MPI_Comm_rank returns in its second argument the calling process' rank in the communicator.

The variable comm_sz is used for the number of processes in MPI_COMM_WORLD, and the variable my_rank for the process rank.

SPMD programs

Most MPI programs are written in a way such that, a single program is written so that different processes carry out different actions, and this is achieved by simply having the processes branch on the basis of their process rank. This approach to parallel programming is called **single program, multiple data**, or **SPMD**.

The **if-else** statement in Lines 16 through 28 makes the program SPMD. The program will, in principle, run with any number of processes.

Communication

In Lines 17 and 18, each process, other than process 0, creates a message it will send to process 0.

Lines 19–20 actually send the message to process 0. Process 0, on the other hand, simply prints its message using printf, and then uses a **for** loop to receive and print the messages sent by processes 1, 2, . . . ,commsz-1.

Lines 24–25 receive the message sent by process q, for $q = 1, 2, \ldots$,commsz-1.

MPI_Send

Each of the sends is carried out by a call to MPI Send,

Syntax

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

The first three arguments**, msg_buf_p, msg_size,** and **msg_type**, determine the contents of the message.

The remaining arguments, **dest, tag**, and **communicator,** determine the destination of the message.

- **msg_buf_p** : pointer to the block of memory containing the contents of the message.
- **msg _size and msg _type**: determine the amount of data to be sent.

- In our program, the msg_size argument is the number of characters in the message plus one character for the '\0'
- The msg_type argument is MPI_CHAR.
- These two arguments together tell the system that the message contains strlen(greeting)+1 chars.

MPI defines a special type, MPI Datatype, that is used for the msg type argument.MPI also defines a number of constant values for this type.

Some predefined MPI Datatypes

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

dest: specifies the rank of the process that should receive the message.

tag: It is a nonnegative int. It can be used to distinguish messages that are otherwise identical.

For example, suppose process 1 is sending floats to process 0. Some of the floats should be printed, while others should be used in a computation. Then the first four arguments to MPI Send provide no information regarding which floats should be printed and which should be used in a computation. So process 1 can

use, say, a tag of 0 for the messages that should be printed and a tag of 1 for the messages that should be used in a computation.

Communicator: All MPI functions that involve communication have a communicator argument. One of the most important purposes of communicators is to specify communication universes. A message sent by a process using one communicator cannot be received by a process that's using a different communicator.

MPI_ Recv

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

The first three arguments specify the memory available for receiving the message.

msg_buf_p: points to the block of memory

buf_size : determines the number of objects that can be stored in the block

buf_type:indicates the type of the objects.

The next three arguments identify the message.

source: specifies the process from which the message should be received.

tag: should match the tag argument of the message being sent

communicator: match the communicator used by the sending process.

Message matching

Suppose process *q* calls MPI Send with

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

Also suppose that process *r* calls MPI_Recv with

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

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

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

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

The status_p argument

A receiver can receive a message without knowing

1. the amount of data in the message

2. the sender of the message, or

3. the tag of the message.

The MPI type MPI_Status is a struct with at least the three members **MPI_ SOURCE, MPI_TAG**, and **MPI_ERROR.**

Suppose our program contains the definition MPI_Status status; Then, after a call to MPI Recv in which &status is passed as the last argument, we can determine the sender and tag by examining the two members

status.MPI SOURCE

status.MPI TAG

The amount of data that's been received isn't stored in a field that's directly accessible to the application program. However, it can be retrieved with a call to MPI_Get_count.

Then the call **MPI_Get_count(&status, recv type, &count)** will return the number of elements received in the count argument.

Syntax of MPI Get count

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

Semantics of MPI Send and MPI Recv

The sending process will assemble the message. For example, it will add the "envelope" information to the actual data being transmitted—the destination process rank, the sending process rank, the tag, the communicator, and some information on the size of the message. Once the message has been assembled, there are essentially two possibilities:

- the sending process can **buffer** the message or

- it can **block**.

If it buffers the message, the MPI system will place the message (data and envelope) into its own internal storage, and the call to MPI Send will return.

If the system blocks, it will wait until it can begin transmitting the message, and the call to MPI Send may not return immediately. Thus, if we use MPI Send, when the function returns, we don't actually know whether the message has been transmitted.

MPI implementations have a default "cutoff" message size. If the size of a message is less than the cutoff, it will be buffered. If the size of the message is greater than the cutoff, MPI Send will block.

MPI Recv always blocks until a matching message has been received. Thus, when a call to MPI Recv returns, we know that there is a message stored in the receive buffer. There is an alternate method for receiving a message, in which the system checks whether a matching message is available and returns, regardless of whether there is one.

MPI requires that messages be **nonovertaking**. This means that if process *q* sends two messages to process *r*, then the first message sent by *q* must be available to *r* before the second message.

However, there is no restriction on the arrival of messages sent from different processes. That is, if *q* and *t* both send messages to *r*,then even if *q* sends its message before *t* sends its message, there is no requirement that*q*'s message become available to *r* before *t*'s message. This is essentially because MPI can't impose performance on a network.

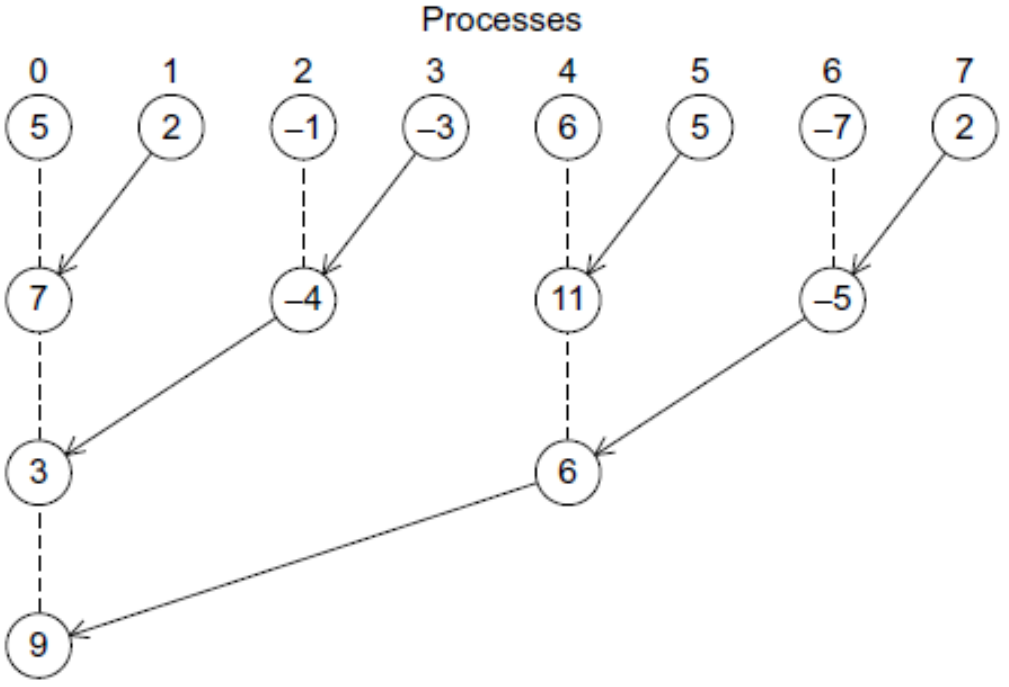
COLLECTIVE COMMUNICATION

Consider the situation to find global sum. Each process with rank greater than 0 is "telling process 0 what to do" and then quitting. That is, each process with rank greater than 0 is, in effect, saying "add this number into the total."

Process 0 is doing nearly all the work in computing the global sum, while the other processes are doing almost nothing.

Tree-structured communication

We might use a "binary tree structure" like that illustrated in Figure below:



In this diagram, initially processes 1, 3, 5, and 7 send their values to processes 0, 2, 4, and 6, respectively. Then processes 0, 2, 4, and 6 add the received values to their original values, and the process is repeated twice:

1. **a.** Processes 2 and 6 send their new values to processes 0 and 4, respectively.

b. Processes 0 and 4 add the received values into their new values.

2. **a.** Process 4 sends its newest value to process 0.

b. Process 0 adds the received value to its newest value.

This solution may not seem ideal, since half the processes (1, 3, 5, and 7) are doing the same amount of work that they did in the original scheme. Tthe original scheme required commsz-1=seven receives and seven adds by process 0, while the new scheme only requires three, and all the other processes do no more than two receives and adds.The new scheme has a property by which a lot of the work is done concurrently by different processes.

For example, in the first phase, the receives and adds by processes 0, 2, 4, and 6 can all take place simultaneously. So, if the processes start at roughly the same time, the total time required to compute the global sum will be the time required by process 0, that is, three receives and three additions. We've thus reduced the overall time by more than 50%.

MPI_Reduce

Communication functions that involve all the processes in a communicator are called **collective communications**.

MPI_Send and MPI_Recv are called **point-to-point** communications.

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

The key to the generalization is the fifth argument, operator. It has type MPI Op, which is a predefined MPI type like MPI_Datatype and MPI_Comm.

The following table shows the different operators:

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

Collective vs. point-to-point communications

1. All the processes in the communicator must call the same collective function. For example, a program that attempts to match a call to MPI Reduce on one process with a call to MPI Recv on another process is erroneous, and, the program will hang or crash.

2. The arguments passed by each process to an MPI collective communication must be "compatible." For example, if one process passes in 0 as the dest process and another passes in 1, then the outcome of a call to MPI Reduce is erroneous, and, once again, the program is likely to hang or crash.

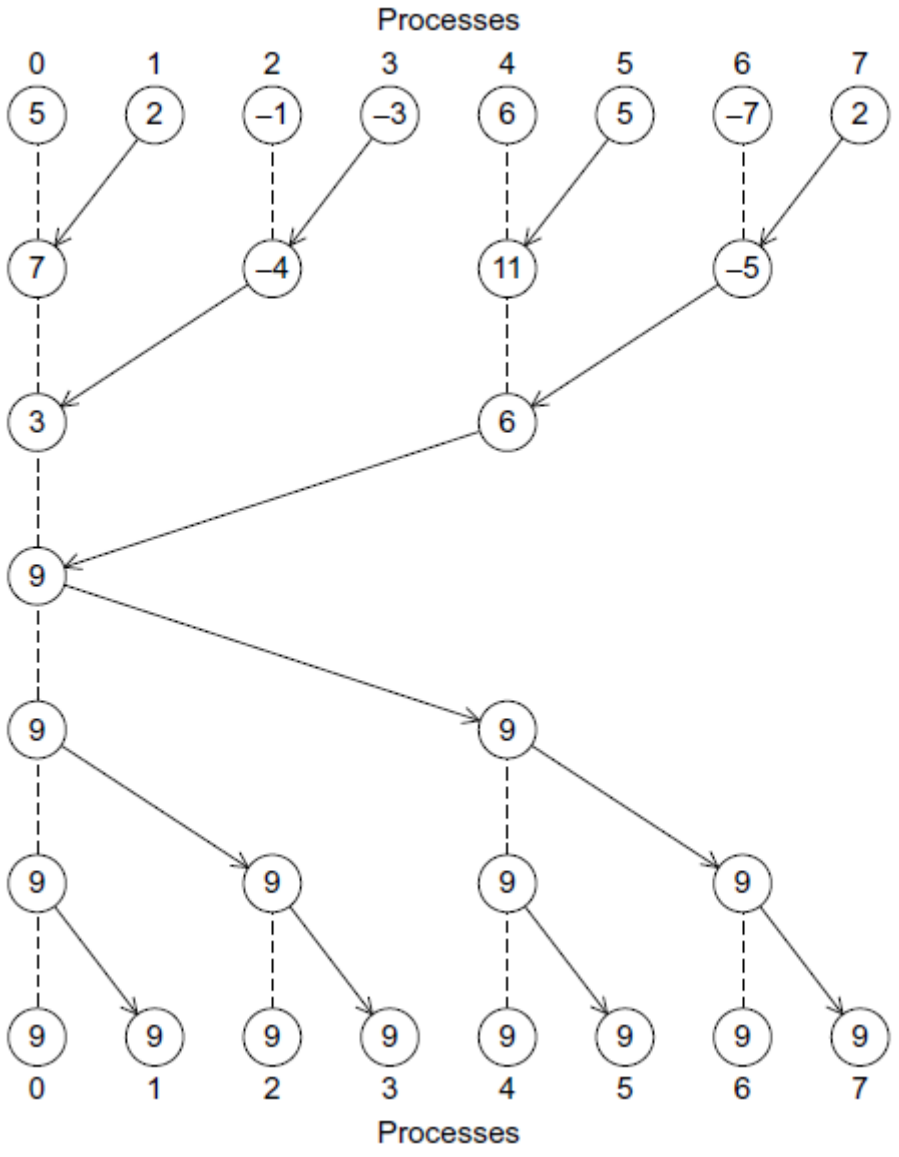
3. The output data p argument is only used on dest process. However, all of the processes still need to pass in an actual argument corresponding to output_data_p, even if it's just NULL.

4. Point-to-point communications are matched on the basis of tags and communicators. Collective communications don't use tags, so they're matched solely on the basis of the communicator and the order in which they're called.

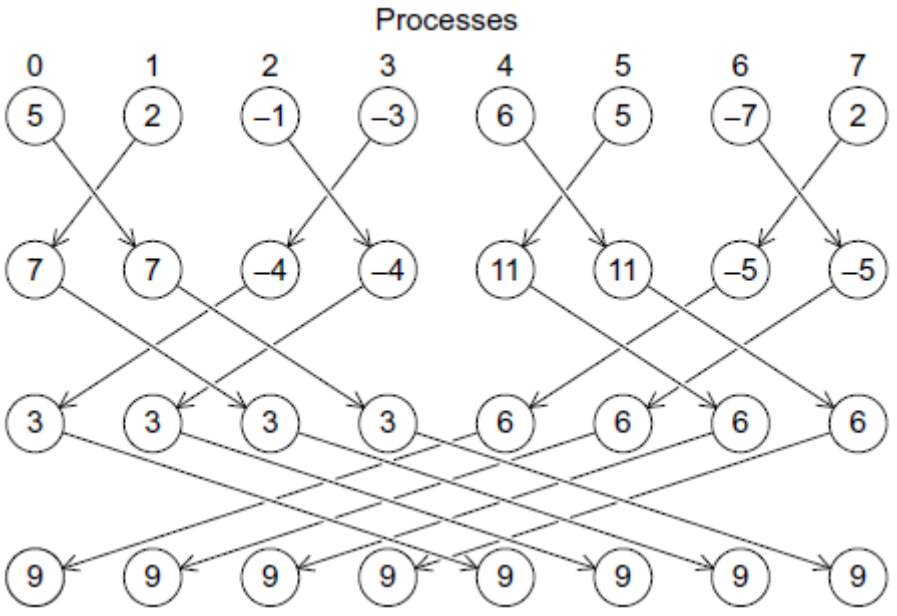
MPI_Allreduce

Consider a situation in which *all* of the processes need the result of a global sum in order to complete some larger computation.

For example, if we use a tree to compute a global sum, we might "reverse" the branches to distribute the global sum.



Alternatively, the processes exchange partial results instead of using one-way communications. Such a communication pattern is sometimes called a **butterfly**.



MPI provides a variant of MPI Reduce that will store the result on all the processes in the communicator:

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

The argument list is identical to that for MPI Reduce except that there is no dest process since all the processes should get the result.

Collective Communication Routines

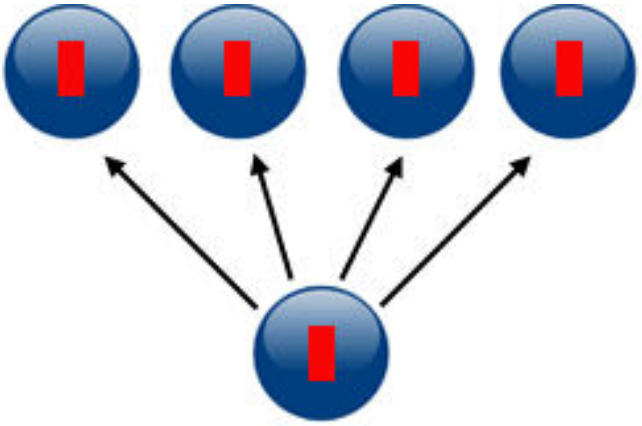
Types of Collective Operations:

- **Synchronization -** processes wait until all members of the group have reached the synchronization point.
- **Data Movement -** broadcast, scatter/gather, all to all.

- **Collective Computation (reductions) -** one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data.

Broadcast

A collective communication in which data belonging to a single process is sent to all of the processes in the communicator is called a **broadcast.**



Broadcast function:

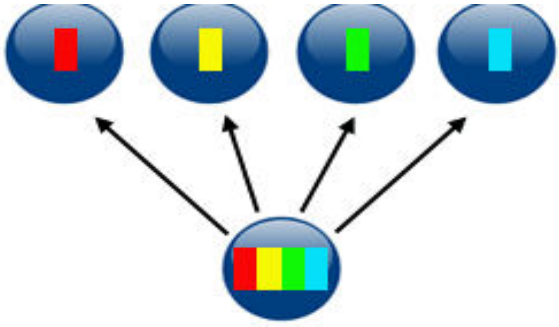
```
int MPI_Bcast(
        void*         data_p          /* in/out */,
        int           count           /* in     */,
        MPI_Datatype  datatype        /* in     */,
        int           source_proc     /* in     */,
        MPI_Comm      comm            /* in     */);
```

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

Data distributions

Scatter

Distributes distinct messages from a single source task to each task in the group.



scatter

Scatter function:

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

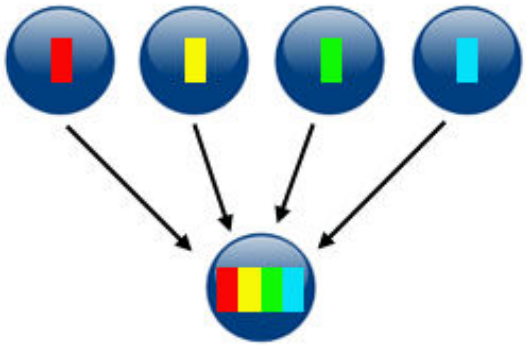
If the communicator comm contains comm_sz processes, then MPI Scatter divides the data referenced by send_buf_p into comm_sz pieces—the first piece goes to process 0,the second to process 1, the third to process 2, and so on.

Example Program:A function for reading and distributing a vector

```
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 */
```

Gather

Gathers distinct messages from each task in the group to a single destination task. This routine is the reverse operation of MPI_Scatter.



Gather function:

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

The data stored in the memory referred to by send_buf_p on process 0 is stored in the first block in recv_buf p, the data stored in the memory referred to by send buf_p on process 1 is stored in the second block referred to by recv_buf_p, and so on. The recv_count is the number of data items received from *each* process, not the total number of data items received.

Example Program:

```
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 */
```

Allgather

Allgather Function:

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

This function concatenates the contents of each process' send_buf_p and stores this in each process' recv_buf_ p. The recv_count is the amount of data being received from *each* process, so in most cases, recv_count will be the same as Send_count.

MPI DERIVED DATATYPES

In MPI, a **derived datatype** can be used to represent any collection of data items in memory by storing both the types of the items and their relative locations in memory.

The idea here is that if a function that sends data knows the types and the relative locations in memory of a collection of data items, it can collect the items from memory before they are sent.

Similarly, a function that receives data can distribute the items into their correct destinations in memory when they're received.

A derived datatype consists of a sequence of basic MPI datatypes together with a *displacement* for each of the datatypes.

For example,suppose that on process 0 the variables a, b, and n are stored in memory locations with the following addresses:

Variable	Address
a	24
b	40
n	48

Then the following derived datatype could represent these data items:

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

The first element of each pair corresponds to the type of the data, and the second element of each pair is the displacement of the data element from the *beginning* of the type.

We've assumed that the type begins with a, so it has displacement 0, and the other elements have displacements measured, in bytes, from a: b is 40-24 =16 bytes beyond the start of a, and n is 48-24 =24 bytes beyond the start of a.

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

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

- **count** : the number of elements in the datatype, so for our example, it should be three.
- **array_of_block_lengths**: Each of the array arguments should have count elements. The first array, array_of_block_lengths, allows for the possibility that the individual data items might be arrays or subarrays. If, for example, the first element were an array containing five elements, we would have array of blocklengths[0] = 5;

  However, in our case, none of the elements is an array, so we can simply define **int array_of_block_lengths[3] ={1, 1, 1};**
- **array_of_displacements:** specifies the displacements, in bytes, from the start of the message. So ,**array_of_displacements[] = {0, 16, 24};**

- **array_of_datatypes**: should store the MPI datatypes of the elements, so **MPI_Datatype array_of_types[3] = {MPI DOUBLE, MPI DOUBLE, MPI INT};**

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

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

PERFORMANCE EVALUATION OF MPI PROGRAMS

Taking timings

MPI provides a function, MPI_Wtime, that returns the number of seconds that have elapsed since some time in the past:

Double MPI_Wtime(**void**);

Thus, we can time a block of MPI code 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);
```

There is a POSIX library function called get timeof day that returns the number of microseconds that have elapsed since some point in the past.There's a C macro GET TIME defined in the header file timer.h.This macro should be called with a **double** argument:

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

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

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

MPI_Barrier

The MPI collective communication function MPI_Barrier insures that no process will return from calling it until every process in the communicator has started calling it.

Syntax

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

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

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

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

Results

As we increase the problem size, the run-times increase, and this is true regardless of the number of processes. As we increase the number of processes, the run-times typically decrease for a while.

The serial run-time is denoted by *T*serial. Since it typically depends on the size of the input, *n*, denoted it as *T*serial(*n*). The parallel run-time is denoted by *T*parallel. Since it depends on both the input size, *n*, and the number of processes, comm._sz= *p*, we'll frequently denote it as *T*parallel(*n*,*p*). The parallel program will divide the work of the serial program among the processes, and add in some overhead time, which we denoted *T*overhead:

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

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

Speedup and efficiency

The most widely used measure of the relation between the serial and the parallel run-times is the **speedup**.

It's just the ratio of the serial run-time to the parallel run-time:

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

The ideal value for *S(n,p)* is *p*. If *S(n,p)*= *p*, then our parallel program with Comm_sz =*p* processes is running *p* times faster than the serial program.

This speedup, sometimes called **linear speedup**. Another widely used measure of parallel performance is parallel **efficiency**. This is "per process" speedup:

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

Scalability

A program is **scalable** if the problem size can be increased at a rate so that the efficiency doesn't decrease as the number of processes increase.

Consider two parallel programs: program *A* and program *B*. Suppose that if$p \geq 2$, the efficiency of program *A* is 0.75, regardless of problem size. Also suppose that the efficiency of program *B* is n=.625p/, provided $p \geq 2$ and $1000 \leq n \leq 625p$.

Then according to our "definition," both programs are scalable.

For program *A*, the rate of increase needed to maintain constant efficiency is 0, while for program *B* if we increase *n* at the same rate as we increase *p*, we'll maintain a constant efficiency.

For example, if $n = 1000$ and $p = 2$, the efficiency of B is 0.80. If we then double p to 4 and we leave the problem size at $n = 1000$, the efficiency will drop to 0.40, but if we also double the problem size to $n = 2000$, the efficiency will remain constant at 0.80. Program A is thus *more* scalable than B, but both satisfy our definition of scalability.

Programs that can maintain a constant efficiency without increasing the problem size are sometimes said to be **strongly scalable**. Programs that can maintain a constant efficiency if the problem size increases at the same rate as the number of processes are sometimes said to be **weakly scalable**.

Program A is strongly scalable, and program B is weakly scalable.

Unit 5

Parallel Program Development

In the last three chapters we haven't just learned about parallel APIs, we've also developed a number of small parallel programs, and each of these programs has involved the implementation of a parallel algorithm. In this chapter, we'll look at a couple of larger examples: solving *n*-body problems and solving the traveling salesperson problem. For each problem, we'll start by looking at a serial solution and examining modifications to the serial solution. As we apply Foster's methodology, we'll see that there are some striking similarities between developing shared- and distributed-memory programs. We'll also see that in parallel programming there are problems that we need to solve for which there is no serial analog. We'll see that there are instances in which, as parallel programmers, we'll have to start "from scratch."

TWO *n*-BODY SOLVERS

In an *n*-body problem, we need to find the positions and velocities of a collection of interacting particles over a period of time. For example, an astrophysicist might want to know the positions and velocities of a collection of stars, while a chemist might want to know the positions and velocities of a collection of molecules or atoms. An *n*-body solver is a program that finds the solution to an *n*-body problem by simulating the behavior of the particles. The input to the problem is the mass, position, and velocity of each particle at the start of the simulation, and the output is typically the position and velocity of each particle at a sequence of user-specified times, or simply the position and velocity of each particle at the end of a user-specified time period.

Let's first develop a serial *n*-body solver. Then we'll try to parallelize it for both shared- and distributed-memory systems.

The problem

For the sake of explicitness, let's write an n-body solver that simulates the motions of planets or stars. We'll use Newton's second law of motion and his law of universal gravitation to determine the positions and velocities. Thus, if particle q has position $\mathbf{s}_q(t)$ at time t, and particle k has position $\mathbf{s}_k(t)$, then the force on particle q exerted by

An Introduction to Parallel Programming. DOI: 10.1016/B978-0-12-374260-5.00006-3

particle k is given by

$$\mathbf{f}_{qk}(t) = -\frac{Gm_q m_k}{|\mathbf{s}_q(t) - \mathbf{s}_k(t)|^3}\left[\mathbf{s}_q(t) - \mathbf{s}_k(t)\right]. \qquad (6.1)$$

Here, G is the gravitational constant ($6.673\times10^{-11}\,m^3/(kg\cdot s^2)$), and m_q and m_k are the masses of particles q and k, respectively. Also, the notation $|\mathbf{s}_q(t) - \mathbf{s}_k(t)|$ represents the distance from particle k to particle q. Note that in general the positions, the velocities, the accelerations, and the forces are vectors, so we're using boldface to represent these variables. We'll use an italic font to represent the other, scalar, variables, such as the time t and the gravitational constant G.

We can use Formula 6.1 to find the total force on any particle by adding the forces due to all the particles. If our n particles are numbered $0,1,2,\ldots,n-1$, then the total force on particle q is given by

$$\mathbf{F}_q(t) = \sum_{\substack{k=0\\k\neq q}}^{n-1} \mathbf{f}_{qk} = -Gm_q \sum_{\substack{k=0\\k\neq q}}^{n-1} \frac{m_k}{|\mathbf{s}_q(t) - \mathbf{s}_k(t)|^3}\left[\mathbf{s}_q(t) - \mathbf{s}_k(t)\right]. \qquad (6.2)$$

Recall that the acceleration of an object is given by the second derivative of its position and that Newton's second law of motion states that the force on an object is given by its mass multiplied by its acceleration, so if the acceleration of particle q is $\mathbf{a}_q(t)$, then $\mathbf{F}_q(t) = m_q\mathbf{a}_q(t) = m_q\mathbf{s}''_q(t)$, where $\mathbf{s}''_q(t)$ is the second derivative of the position $\mathbf{s}_q(t)$. Thus, we can use Formula 6.2 to find the acceleration of particle q:

$$\mathbf{s}''_q(t) = -G\sum_{\substack{j=0\\j\neq q}}^{n-1} \frac{m_j}{|\mathbf{s}_q(t) - \mathbf{s}_j(t)|^3}\left[\mathbf{s}_q(t) - \mathbf{s}_j(t)\right]. \qquad (6.3)$$

Thus Newton's laws give us a system of *differential* equations—equations involving derivatives—and our job is to find at each time t of interest the position $\mathbf{s}_q(t)$ and velocity $\mathbf{v}_q(t) = \mathbf{s}0_q(t)$.

We'll suppose that we either want to find the positions and velocities at the times

$t = 0, 1t, 2 1t, \ldots, T1t,$

or, more often, simply the positions and velocities at the final time $T1t$. Here, $1t$ and T are specified by the user, so the input to the program will be n, the number of particles, $1t$, T, and, for each particle, its mass, its initial position, and its initial velocity. In a fully general solver, the positions and velocities would be three-dimensional vectors, but in order to keep things simple, we'll assume that the particles will move in a plane, and we'll use two-dimensional vectors instead.

The output of the program will be the positions and velocities of the n particles at the timesteps $0, 1t, 2 1t, \ldots$, or just the positions and velocities at $T1t$. To get the output at only the final time, we can add an input option in which the user specifies whether she only wants the final positions and velocities.

Two serial programs

In outline, a serial n-body solver can be based on the following pseudocode:

```
1    Get input data;
2    for each timestep {
3    if (timestep output) Print positions and velocities of particles;
4    for each particle q
5    Compute total force on q;
6    for each particle q
7    Compute position and velocity of q;
8    }
9    Print positions and velocities of particles;
```

We can use our formula for the total force on a particle (Formula 6.2) to refine our pseudocode for the computation of the forces in Lines 4 – 5:

```
for each particle q {
    for each particle k != q {
        x_diff = pos[q][X] − pos[k][X]; y_diff = pos[q][Y] − pos[k][Y]; dist = sqrt(x_diff∗x_diff + y_diff∗y_diff);
        dist_cubed = dist∗dist∗dist; forces[q][X] −= G∗masses[q]∗masses[k]/dist_cubed ∗ x_diff; forces[q][Y]
        −= G∗masses[q]∗masses[k]/dist_cubed ∗ y_diff;
    }
}
```

Here, we're assuming that the forces and the positions of the particles are stored as two-dimensional arrays, forces and pos, respectively. We're also assuming we've defined constants X = 0 and Y = 1. So the x-component of the force on particle q is forces[q][X] and the y-component is forces[q][Y]. Similarly, the components of the position are pos[q][X] and pos[q][Y]. (We'll take a closer look at data structures shortly.)

We can use Newton's third law of motion, that is, for every action there is an equal and opposite reaction, to halve the total number of calculations required for the forces. If the force on particle q due to particle k is $\mathbf{f}_{qk}$, then the force on k due to q is $-\mathbf{f}_{qk}$. Using this simplification we can modify our code to compute forces, as shown in Program 6.1. To better understand this pseudocode, imagine the individual forces as a two-dimensional array:

$$\begin{bmatrix} 0 & \mathbf{f}_{01} & \mathbf{f} & \cdots & \mathbf{f}_{0,n-1} \\ & 0 & \mathbf{f} & & \end{bmatrix}$$

$$-\mathbf{f}_{01} \quad \cdots \quad \mathbf{f}_{1,n-1}$$

$$\begin{bmatrix} -\mathbf{f}_{02} & -\mathbf{f}_{12} & 0 & \cdots & \mathbf{f}_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -\mathbf{f}_{0,n-1} & -\mathbf{f}_{1,n-1} & -\mathbf{f}_{2,n-1} & \cdots & 0 \end{bmatrix}.$$

(Why are the diagonal entries 0?) Our original solver simply adds all of the entries in row q to get forces[q]. In our modified solver, when $q = 0$, the body of the loop

```
for  each  particle  q
     forces[q]  =  0;
for  each  particle  q  {
     for  each  particle  k  >  q  {
          x_diff  =  pos[q][X]  -  pos[k][X];
          y_diff  =  pos[q][Y]  -  pos[k][Y];
          dist  =  sqrt(x_diff  *x_diff  +  y_diff  *y_diff);
          dist_cubed  =  dist  *dist  *dist;
          force_qk[X]  =  G*masses[q]  *masses[k]/dist_cubed  *  x_diff;
          force_qk[Y]  =  G*masses[q]  *masses[k]/dist_cubed  *  y_diff

          forces[q][X]  +=  force_qk[X];
          forces[q][Y]  +=  force_qk[Y];
          forces[k][X]  -=  force_qk[X];
          forces[k][Y]  -=  force_qk[Y];
     }
}
```

Program 6.1: A reduced algorithm for computing *n*-body forces

foreach particle q will add the entries in row 0 into forces[0]. It will also add the *k*th entry in column 0 into forces[k] for $k = 1,2,\ldots,n-1$. In general, the *q*th iteration will add the entries to the right of the diagonal (that is, to the right of the 0) in row *q* into forces[q], and the entries below the diagonal in column *q* will be added into their respective forces, that is, the *k*th entry will be added in to forces[k].

Note that in using this modified solver, it's necessary to initialize the forces array in a separate loop, since the *q*th iteration of the loop that calculates the forces will, in general, add the values it computes into forces[k] for $k = q+1,q+2,\ldots, n-1$, not just forces[q].

In order to distinguish between the two algorithms, we'll call the *n*-body solver with the original force calculation, the *basic* algorithm, and the solver with the number of calculations reduced, the *reduced* algorithm.

The position and the velocity remain to be found. We know that the acceleration of particle *q* is given by

$$\mathbf{a}_q(t) = \mathbf{s}_q''(t) = \mathbf{F}_q(t)/m_q,$$

where $\mathbf{s}00_q(t)$ is the second derivative of the position $\mathbf{s}_q(t)$ and $\mathbf{F}_q(t)$ is the force on particle *q*. We also know that the velocity $\mathbf{v}_q(t)$ is the first derivative of the position $\mathbf{s}0_q(t)$, so we need to integrate the acceleration to get the velocity, and we need to integrate the velocity to get the position.

We might at first think that we can simply find an antiderivative of the function in Formula 6.3. However, a second look shows us that this approach has problems: the right-hand side contains unknown functions $\mathbf{s}_q$ and $\mathbf{s}_k$—not just the variable *t*—so we'll instead use a **numerical** method for *estimating* the position and the velocity. This means that rather than trying to find simple closed formulas, we'll approximate

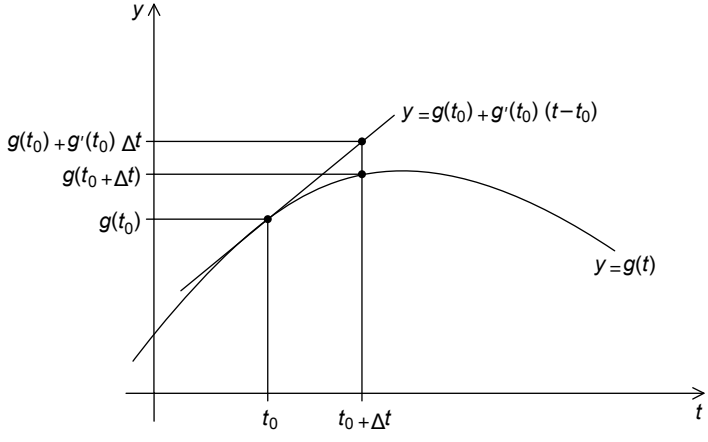


FIGURE 6.1

Using the tangent line to approximate a function

the values of the position and velocity at the times of interest. There are *many* possible choices for numerical methods, but we'll use the simplest one: Euler's method, which is named after the famous Swiss mathematician Leonhard Euler (1707–1783). In Euler's method, we use the tangent line to approximate a function. The basic idea is that if we know the value of a function $g(t_0)$ at time t_0 and we also know its derivative $g'(t_0)$ at time t_0, then we can approximate its value at time $t_0 + \Delta t$ by using the tangent line to the graph of $g(t_0)$. See Figure 6.1 for an example. Now if we know a point $(t_0, g(t_0))$ on a line, and we know the slope of the line $g'(t_0)$, then an equation for the line is given by

$$y = g(t_0) + g'(t_0)(t - t_0).$$

Since we're interested in the time $t = t_0 + \Delta t$, we get

$$g(t + \Delta t) \approx g(t_0) + g'(t_0)(t + \Delta t - t) = g(t_0) + \Delta t\, g'(t_0).$$

Note that this formula will work even when $g(t)$ and y are vectors: when this is the case, $g'(t)$ is also a vector and the formula just adds a vector to a vector multiplied by a scalar, Δt.

Now we know the value of $\mathbf{s}_q(t)$ and $\mathbf{s}'_q(t)$ at time 0, so we can use the tangent line and our formula for the acceleration to compute $\mathbf{s}_q(\Delta t)$ and $\mathbf{v}_q(\Delta t)$:

$$\mathbf{s}_q(\Delta t) \approx \mathbf{s}_q(0) + \Delta t\, \mathbf{s}'_q(0) = \mathbf{s}_q(0) + \Delta t\, \mathbf{v}_q(0),$$

$$(\Delta t) \approx \mathbf{v}_q(0) + \Delta t\, \mathbf{v}'_q(0) = \mathbf{v}_q(0) + \Delta t\, \mathbf{a}_q(0) = \mathbf{v}_q(0) + \Delta \qquad \frac{1}{m_q} \mathbf{v}_q t\, \mathbf{F}_q(0).$$

When we try to extend this approach to the computation of $\mathbf{s}_q(2\Delta t)$ and $\mathbf{s}'_q(2\Delta t)$, we see that things are a little bit different, since we don't know the exact value of $\mathbf{s}_q(\Delta t)$

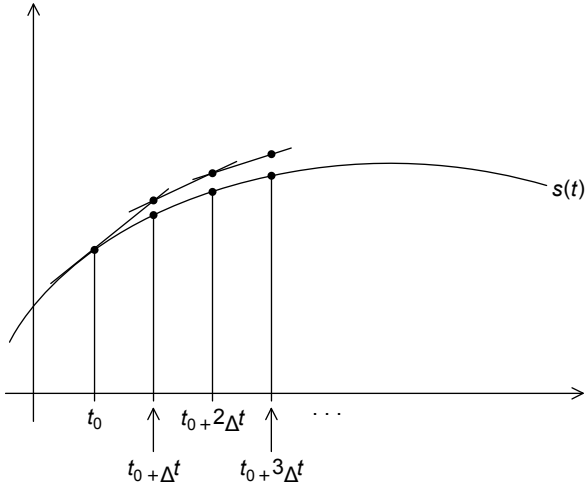


FIGURE 6.2

Euler's method

and $\mathbf{s}0_q(1t)$. However, if our approximations to $\mathbf{s}_q(1t)$ and $\mathbf{s}'_q(\Delta t)$ are good, then we should be able to get a reasonably good approximation to $\mathbf{s}_q(21t)$ and $\mathbf{s}0_q(21t)$ using the same idea. This is what Euler's method does (see Figure 6.2).

Now we can complete our pseudocode for the two n-body solvers by adding in the code for computing position and velocity:

```
pos[q][X]      +=  -  deltat*vel[q][X];      pos[q][Y]      +=
deltat*vel[q][Y];      -  vel[q][X]  +=  deltat/masses[q]*forces[q][X];
vel[q][Y]      +=  -  deltat/masses[q]*forces[q][Y];
                   -
```

Here, we're using `pos[q]`, `vel[q]`, and `forces[q]` to store the position, the velocity, and the force, respectively, of particle q.

Before moving on to parallelizing our serial program, let's take a moment to look at data structures. We've been using an array type to store our vectors:

```
#define DIM 2 typedef double vect t[DIM];
```

A struct is also an option. However, if we're using arrays and we decide to change our program so that it solves three-dimensional problems, in principle, we only need to change the macro `DIM`. If we try to do this with structs, we'll need to rewrite the code that accesses individual components of the vector.

For each particle, we need to know the values of

- its mass,
- its position,
- its velocity,
- its acceleration, and
- the total force acting on it.

Since we're using Newtonian physics, the mass of each particle is constant, but the other values will, in general, change as the program proceeds. If we examine our code, we'll see that once we've computed a new value for one of these variables for a given timestep, we never need the old value again. For example, we don't need to do anything like this

```
new pos q = f(old pos q); new vel q = g(old pos q, new pos q);
```

Also, the acceleration is only used to compute the velocity, and its value can be computed in one arithmetic operation from the total force, so we only need to use a local, temporary variable for the acceleration.

For each particle it suffices to store its mass and the current value of its position, velocity, and force. We could store these four variables as a struct and use an array of structs to store the data for all the particles. Of course, there's no reason that all of the variables associated with a particle need to be grouped together in a struct. We can split the data into separate arrays in a variety of different ways. We've chosen to group the mass, position, and velocity into a single struct and store the forces in a separate array. With the forces stored in contiguous memory, we can use a fast function such as memset to quickly assign zeroes to all of the elements at the beginning of each iteration:

```
#include <string.h> /* For memset */
. . .
vect t* forces = malloc(n*sizeof(vect t));
. . .
for (step = 1; step <= n steps; step++) { . . .
   /* Assign 0 to each element of the forces array */ forces = memset(forces, 0,
   n*sizeof(vect t); for (part = 0; part < n-1; part++) Computeforce(part, forces, . . .)
   . . .
}
```

If the force on each particle were a member of a struct, the force members wouldn't occupy contiguous memory in an array of structs, and we'd have to use a relatively slow **for** loop to assign zero to each element.

Parallelizing the *n*-body solvers

Let's try to apply Foster's methodology to the *n*-body solver. Since we initially want *lots* of tasks, we can start by making our tasks the computations of the positions, the velocities, and the total forces at each timestep. In the basic algorithm, the algorithm in which the total force on each particle is calculated directly from Formula 6.2, the

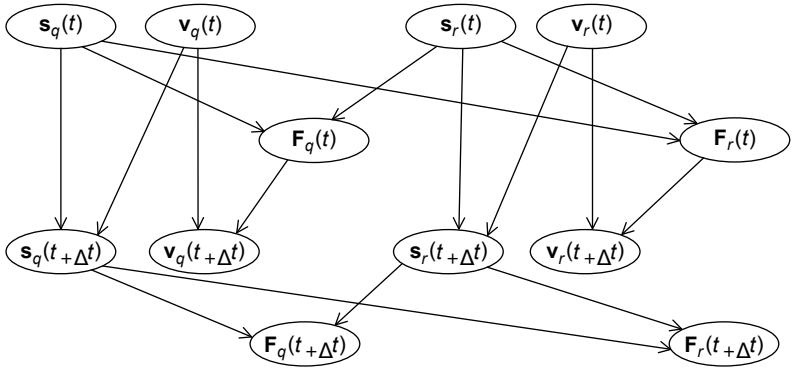


FIGURE 6.3

Communications among tasks in the basic n-body solver

computation of $\mathbf{F}_q(t)$, the total force on particle q at time t, requires the positions of each of the particles $\mathbf{s}_r(t)$, for each r. The computation of $\mathbf{v}_q(t+1t)$ requires the velocity at the previous timestep, $\mathbf{v}_q(t)$, and the force, $\mathbf{F}_q(t)$, at the previous timestep. Finally, the computation of $\mathbf{s}_q(t+1t)$ requires $\mathbf{s}_q(t)$ and $\mathbf{v}_q(t)$. The communications among the tasks can be illustrated as shown in Figure 6.3. The figure makes it clear that most of the communication among the tasks occurs among the tasks associated with an individual particle, so if we agglomerate the computations of $\mathbf{s}_q(t)$,$\mathbf{v}_q(t)$, and $\mathbf{F}_q(t)$, our intertask communication is greatly simplified (see Figure 6.4). Now the tasks correspond to the particles and, in the figure, we've labeled the communications with the data that's being communicated. For example, the arrow from particle q at timestep t to particle r at timestep t is labeled with $\mathbf{s}_q$, the position of particle q.

For the reduced algorithm, the "intra-particle" communications are the same. That is, to compute $\mathbf{s}_q(t+1t)$ we'll need $\mathbf{s}_q(t)$ and $\mathbf{v}_q(t)$, and to compute $\mathbf{v}_q(t+1t)$, we'll need $\mathbf{v}_q(t)$ and $\mathbf{F}_q(t)$. Therefore, once again it makes sense to agglomerate the computations associated with a single particle into a composite task.

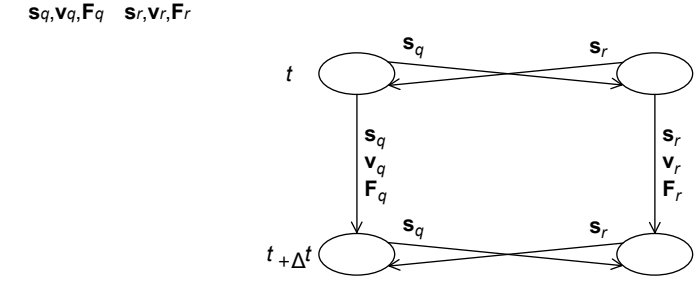


FIGURE 6.4

Communications among agglomerated tasks in the basic n-body solver

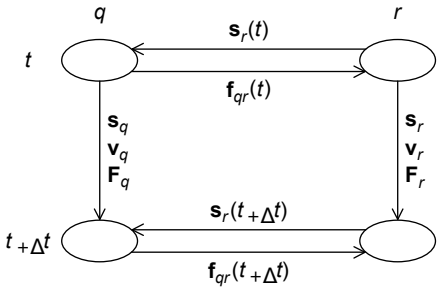


FIGURE 6.5

Communications among agglomerated tasks in the reduced *n*-body solver *(q<r)*

Recollect that in the reduced algorithm, we make use of the fact that the force $\mathbf{f}_{rq} = -\mathbf{f}_{qr}$. So if $q < r$, then the communication *from* task *r to* task *q* is the same as in the basic algorithm—in order to compute $\mathbf{F}_q(t)$, task/particle *q* will need $\mathbf{s}_r(t)$ from task/particle *r*. However, the communication from task *q* to task *r* is no longer $s_q(t)$, it's the force on particle *q* due to particle *r*, that is, $\mathbf{f}_{qr}(t)$. See Figure 6.5.

The final stage in Foster's methodology is mapping. If we have *n* particles and *T* timesteps, then there will be *nT* tasks in both the basic and the reduced algorithm. Astrophysical *n*-body problems typically involve thousands or even millions of particles, so *n* is likely to be several orders of magnitude greater than the number of available cores. However, *T* may also be much larger than the number of available cores. So, in principle, we have two "dimensions" to work with when we map tasks to cores. However, if we consider the nature of Euler's method, we'll see that attempting to assign tasks associated with a single particle at different timesteps to different cores won't work very well. Before estimating $\mathbf{s}_q(t+1t)$ and $\mathbf{v}_q(t+1t)$, Euler's method must "know" $\mathbf{s}_q(t)$, $\mathbf{v}_q(t)$, and $\mathbf{a}_q(t)$. Thus, if we assign particle *q* at time *t* to core c_0, and we assign particle *q* at time $t+1t$ to core c_1 6= c_0, then we'll have to communicate $\mathbf{s}_q(t)$,$\mathbf{v}_q(t)$, and $\mathbf{F}_q(t)$ from c_0 to c_1. Of course, if particle *q* at time *t* and particle *q* at time $t+1t$ are mapped to the same core, this communication won't be necessary, so once we've mapped the task consisting of the calculations for particle *q* at the first timestep to core c_0, we may as well map the subsequent computations for particle *q* to the same cores, since we can't simultaneously execute the computations for particle *q* at two different timesteps. Thus, mapping tasks to cores will, in effect, be an assignment of particles to cores.

At first glance, it might seem that any assignment of particles to cores that assigns roughly *n*/threadcount particles to each core will do a good job of balancing the workload among the cores, and for the basic algorithm this is the case. In the basic algorithm the work required to compute the position, velocity, and force is the same for every particle. However, in the reduced algorithm the work required in the forces computation loop is much greater for lower-numbered iterations than the work required for higher-numbered iterations. To see this, recall the pseudocode that computes the total force on particle *q* in the reduced algorithm:

```
for each particle k > q {
    x_diff = pos[q][X] − pos[k][X]; y_diff = pos[q][Y] − pos[k][Y]; dist = sqrt(x_diff∗x_diff + y_diff∗y_diff);
    distcubed = dist∗dist∗dist; forceqk[X] = G∗masses[q]∗masses[k]/distcubed ∗ x_diff;
    forceqk[Y] = G∗masses[q]∗masses[k]/distcubed ∗ y_diff;

    forces[q][X] += forceqk[X];   forces[q][Y] += forceqk[Y];
    forces[k][X] −= forceqk[X];   forces[k][Y] −= forceqk[Y];
}
```

Then, for example, when $q = 0$, we'll make $n-1$ passes through the **for** each particle k > q loop, while when $q = n-1$, we won't make any passes through the loop. Thus, for the reduced algorithm we would expect that a cyclic partition of the particles would do a better job than a block partitition of evenly distributing the *computation*.

However, in a shared-memory setting, a cyclic partition of the particles among the cores is almost certain to result in a much higher number of cache misses than a block partition, and in a distributed-memory setting, the overhead involved in communicating data that has a cyclic distribution will probably be greater than the overhead involved in communicating data that has a block distribution (see Exercises 6.8 and 6.9).

Therefore with a composite task consisting of all of the computations associated with a single particle throughout the simulation, we conclude the following:

1. A block distribution will give the best performance for the basic *n*-body solver.
2. For the reduced *n*-body solver, a cyclic distribution will best distribute the workload in the computation of the forces. However, this improved performance *may* be offset by the cost of reduced cache performance in a shared-memory setting and additional communication overhead in a distributed-memory setting.

In order to make a final determination of the optimal mapping of tasks to cores, we'll need to do some experimentation.

A word about I/O

You may have noticed that our discussion of parallelizing the *n*-body solver hasn't touched on the issue of I/O, even though I/O can figure prominently in both of our serial algorithms. We've discussed the problem of I/O several times in earlier chapters. Recall that different parallel systems vary widely in their I/O capabilities, and with the very basic I/O that is commonly available it is very difficult to obtain high performance. This basic I/O was designed for use by single-process, single-threaded programs, and when multiple processes or multiple threads attempt to access the I/O buffers, the system makes no attempt to schedule their access. For example, if multiple threads attempt to execute printf("Hello from thread %d of %d\n", myrank, threadcount);

more or less simultaneously, the order in which the output appears will be unpredictable. Even worse, one thread's output may not even appear as a single line. It can happen that the output from one thread appears as multiple segments, and the individual segments are separated by output from other threads.

Thus, as we've noted earlier, except for debug output, we generally assume that one process/thread does all the I/O, and when we're timing program execution, we'll use the option to only print output for the final timestep. Furthermore, we won't include this output in the reported run-times.

Of course, even if we're ignoring the cost of I/O, we can't ignore its existence. We'll briefly discuss its implementation when we discuss the details of our parallel implementations.

Parallelizing the basic solver using OpenMP

How can we use OpenMP to map tasks/particles to cores in the basic version of our *n*-body solver? Let's take a look at the pseudocode for the serial program:

```
for each timestep { if (timestep output) Print positions and velocities of particles; for each particle q
        Compute total force on q; for each particle q
        Compute position and velocity of q;
}
```

The two inner loops are both iterating over particles. So, in principle, parallelizing the two inner **for** loops will map tasks/particles to cores, and we might try something like this:

```
    for each timestep { if (timestep output) Print positions and velocities of particles;
#       pragma omp parallel for for each particle q
            Compute total force on q;
#       pragma omp parallel for for each particle q
            Compute position and velocity of q;
    }
```

We may not like the fact that this code could do a lot of forking and joining of threads, but before dealing with that, let's take a look at the loops themselves: we need to see if there are any race conditions caused by loop-carried dependences.

In the basic version the first loop has the following form:

```
# pragma omp parallel for for each particle q {
        forces[q][X] = forces[q][Y] = 0 ; for each particle k != q {
            xdiff = pos[q][X] − pos[k][X]; y_diff = pos[q][Y] − pos[k][Y];
            dist = sqrt(x diff∗x diff + y diff∗y diff); distcubed =
            dist∗dist∗dist;                forces[q][X]                −=
            G∗masses[q]∗masses[k]/distcubed ∗ xdiff; forces[q][Y] −=
            G∗masses[q]∗masses[k]/dist_cubed ∗ y_diff;
```

```
        }
    }
```

Since the iterations of the **for** each particle q loop are partitioned among the threads, only one thread will access forces[q] for any q. Different threads do access the same elements of the pos array and the masses array. However, these arrays are only *read* in the loop. The remaining variables are used for temporary storage in a single iteration of the inner loop, and they can be private. Thus, the parallelization of the first loop in the basic algorithm won't introduce any race conditions.

The second loop has the form:

```
# pragma omp parallel for for each particle q {
        pos[q][X]  +=  deltat*vel[q][X];   pos[q][Y]  +=
        deltat*vel[q][Y];   vel[q][X]  +=  deltat/masses[q]*forces[q][X];
        vel[q][Y]  +=  deltat/masses[q]*forces[q][Y];
    }
```

Here, a single thread accesses pos[q], vel[q], masses[q], and forces[q] for any particle q, and the scalar variables are only read, so parallelizing this loop also won't introduce any race conditions.

Let's return to the issue of repeated forking and joining of threads. In our pseudocode, we have

```
    for each timestep { if (timestep output) Print positions and velocities of particles;
#       pragma omp parallel for for each particle q
            Compute total force on q;
#       pragma omp parallel for for each particle q
            Compute position and velocity of q;
    }
```

We encountered a similar issue when we parallelized odd-even transposition sort (see Section 5.6.2). In that case, we put a parallel directive before the outermost loop and used OpenMP **for** directives for the inner loops. Will a similar strategy work here? That is, can we do something like this?

```
# pragma omp parallel for each timestep { if (timestep output) Print positions and velocities of
    particles;
#     pragma omp for
        for each particle q
            Compute total force on q;
#     pragma omp for
        for each particle q
            Compute position and velocity of q;
```

```
    }
```

This will have the desired effect on the two **for** each particle loops: the same team of threads will be used in both loops and for every iteration of the outer loop. However, we have a clear problem with the output statement. As it stands now, every thread will print all the positions and velocities, and we only want one thread to do the I/O. However, OpenMP provides the single directive for exactly this situation: we have a team of threads executing a block of code, but a part of the code should only be executed by one of the threads. Adding the single directive gives us the following pseudocode:

```
# pragma omp parallel for each timestep {
        if (timestep output) {
#       pragma omp single
          Print positions and velocities of particles;
#         }
          pragma omp for for each
          particle q
              Compute total force on q;
#       pragma omp for
          for each particle q
              Compute position and velocity of q;
    }
```

There are still a few issues that we need to address. The most important has to do with possible race conditions introduced in the transition from one statement to another. For example, suppose thread 0 completes the first **for**each particle loop before thread 1, and it then starts updating the positions and velocities of its assigned particles in the second **for** each particle loop. Clearly, this could cause thread 1 to use an updated position in the first **for**each particle loop. However, recall that there is an implicit barrier at the end of each structured block that has been parallelized with a **for** directive. So, if thread 0 finishes the first inner loop before thread 1, it will block until thread 1 (and any other threads) finish the first inner loop, and it won't start the second inner loop until all the threads have finished the first. This will also prevent the possibility that a thread might rush ahead and print positions and velocities before they've all been updated by the second loop.

There's also an implicit barrier after the single directive, although in this program the barrier isn't necessary. Since the output statement won't update any memory locations, it's OK for some threads to go ahead and start executing the next iteration before output has been completed. Furthermore, the first inner **for** loop in the next iteration only updates the forces array, so it can't cause a thread executing the output statement to print incorrect values, and because of the barrier at the end of the first inner loop, no thread can race ahead and start updating positions and velocities in the second inner loop before the output has been completed. Thus, we could modify the single directive with a nowait clause. If the OpenMP implementation supports it, this simply eliminates the implied barrier associated with the single directive. It can also be used with **for**, parallel **for**, and parallel directives. Note that in this case, addition of the nowait clause is unlikely to have much effect on performance, since the two **for**each particle loops have implied barriers that will prevent any one thread from getting more than a few statements ahead of any other.

Finally, we may want to add a schedule clause to each of the **for** directives in order to insure that the iterations have a block partition:

```
#       pragma omp for schedule(static, n/threadcount)
```

Parallelizing the reduced solver using OpenMP

The reduced solver has an additional inner loop: the initialization of the forces array to 0. If we try to use the same parallelization for the reduced solver, we should also parallelize this loop with a **for** directive. What happens if we try this? That is, what happens if we try to parallelize the reduced solver with the following pseudocode?

```
# pragma omp parallel for each timestep {
        if (timestep output) {
#       pragma omp single
          Print positions and velocities of particles;
#         }
          pragma omp for for each
          particle q forces[q] = 0.0 ;

#         pragma omp for for each
          particle q
              Compute total force on q;
#       pragma omp for
          for each particle q
              Compute position and velocity of q;
    }
```

Parallelization of the initialization of the forces should be fine, as there's no dependence among the iterations. The updating of the positions and velocities is the same in both the basic and reduced solvers, so if the computation of the forces is OK, then this should also be OK.

How does parallelization affect the correctness of the loop for computing the forces? Recall that in the reduced version, this loop has the following form:

```
# pragma omp for /* Can be faster than memset */ for each particle q {
        forceqk[X] = forceqk[Y] = 0 ; for each particle k > q {

            x diff = pos[q][X] − pos[k][X]; y diff = pos[q][Y] − pos[k][Y]; dist = sqrt(x diff*x diff + y diff*y
            diff); distcubed = dist*dist*dist; forceqk[X] = G*masses[q]*masses[k]/distcubed * xdiff;
            forceqk[Y] = G*masses[q]*masses[k]/distcubed * y_diff;
```

```
            forces[q][X] += forceqk[X];  forces[q][Y] += forceqk[Y];
            forces[k][X] -= forceqk[X];  forces[k][Y] -= forceqk[Y];
        }
    }
```

As before, the variables of interest are pos, masses, and forces, since the values in the remaining variables are only used in a single iteration, and hence, can be private. Also, as before, elements of the pos and masses arrays are only read, not updated. We therefore need to look at the elements of the forces array. In this version, unlike the basic version, a thread *may* update elements of the forces array other than those corresponding to its assigned particles. For example, suppose we have two threads and four particles and we're using a block partition of the particles. Then the total force on particle 3 is given by

$$\mathbf{F}_3 = -\mathbf{f}_{03} - \mathbf{f}_{13} - \mathbf{f}_{23}.$$

Furthermore, thread 0 will compute $\mathbf{f}_{03}$ and $\mathbf{f}_{13}$, while thread 1 will compute $\mathbf{f}_{23}$. Thus, the updates to forces[3] *do* create a race condition. In general, then, the updates to the elements of the forces array introduce race conditions into the code.

A seemingly obvious solution to this problem is to use a critical directive to limit access to the elements of the forces array. There are at least a couple of ways to do this. The simplest is to put a critical directive before all the updates to forces

```
# pragma omp critical
    {
        forces[q][X] += forceqk[X];  forces[q][Y] += forceqk[Y];
        forces[k][X] -= forceqk[X];  forces[k][Y] -= forceqk[Y];
    }
```

However, with this approach access to the elements of the forces array will be effectively serialized. Only one element of forces can be updated at a time, and contention for access to the critical section is actually likely to seriously degrade the performance of the program. See Exercise 6.3.

An alternative would be to have one critical section for each particle. However, as we've seen, OpenMP doesn't readily support varying numbers of critical sections, so we would need to use one lock for each particle instead and our updates would look something like this:

```
omp_set_lock(&locks[q]); forces[q][X] += forceqk[X];
forces[q][Y] += force_qk[Y]; omp unset
lock(&locks[q]);

omp_set_lock(&locks[k]); forces[k][X] -= forceqk[X];
forces[k][Y] -= force_qk[Y]; omp unset
lock(&locks[k]);
```

This assumes that the master thread will create a shared array of locks, one for each particle, and when we update an element of the forces array, we first set the lock corresponding to that particle. Although this approach performs much better than the single critical section, it still isn't competitive with the serial code. See Exercise 6.4.

Another possible solution is to carry out the computation of the forces in two phases. In the first phase, each thread carries out exactly the same calculations it carried out in the erroneous parallelization. However, now the calculations are stored in its *own* array of forces. Then, in the second phase, the thread that has been assigned particle q will add the contributions that have been computed by the different threads. In our example above, thread 0 would compute $-\mathbf{f}_{03}$ $-\mathbf{f}_{13}$, while thread 1 would compute $-\mathbf{f}_{23}$. After each thread was done computing its contributions to the forces, thread 1, which has been assigned particle 3, would find the total force on particle 3 by adding these two values.

Let's look at a slightly larger example. Suppose we have three threads and six particles. If we're using a block partition of the particles, then the computations in the first phase are shown in Table 6.1. The last three columns of the table show each thread's contribution to the computation of the total forces. In phase 2 of the computation, the thread specified in the first column of the table will add the contents of each of its assigned rows—that is, each of its assigned particles.

Note that there's nothing special about using a block partition of the particles. Table 6.2 shows the same computations if we use a cyclic partition of the particles.

Table 6.1 First-Phase Computations for a Reduced Algorithm with Block Partition

		Thread		
Thread	**Particle**	**0**	**1**	**2**
	0	0 $\mathbf{f}_{01}+\mathbf{f}_{02}+\mathbf{f}_{03}+\mathbf{f}_{04}+\mathbf{f}_{05}$	0	0
	1	$-\mathbf{f}_{01}+\mathbf{f}_{12}+\mathbf{f}_{13}+\mathbf{f}_{14}+\mathbf{f}_{15}$	0	0
1	2	$-\mathbf{f}_{02}$ $-\mathbf{f}_{12}$ $\mathbf{f}_{23}+\mathbf{f}_{24}+\mathbf{f}_{25}$		0
3		f $-03-\mathbf{f}_{13}$ $-\mathbf{f}_{23}+\mathbf{f}_{34}+\mathbf{f}_{35}$		0
2	4	$-\mathbf{f}_{04}-\mathbf{f}_{14}$ $-\mathbf{f}_{24}-\mathbf{f}_{34}$		$\mathbf{f}_{45}$
	5	$-\mathbf{f}_{05}-\mathbf{f}_{15}$ $-\mathbf{f}_{25}-\mathbf{f}_{35}$		$-\mathbf{f}_{45}$

Table 6.2 First-Phase Computations for a Reduced Algorithm with Cyclic Partition

		Thread		
Thread	**Particle**	**0**	**1**	**2**
0 0	$\mathbf{f}_{01}+\mathbf{f}_{02}+\mathbf{f}_{03}+\mathbf{f}_{04}+\mathbf{f}_{05}$	0 0 1 1	$-\mathbf{f}_{01}$ $\mathbf{f}_{12}+\mathbf{f}_{13}+\mathbf{f}_{14}+\mathbf{f}_{15}$	0
2	2	$-\mathbf{f}_{02}$	$-\mathbf{f}_{12}$ $\mathbf{f}_{23}+\mathbf{f}_{24}+\mathbf{f}_{25}$	
0	3	$-\mathbf{f}_{03}+\mathbf{f}_{34}+\mathbf{f}_{35}$	$-\mathbf{f}_{13}$ $-\mathbf{f}_{23}$	
1	4	$-\mathbf{f}_{04}-\mathbf{f}_{34}$	$-\mathbf{f}_{14}+\mathbf{f}_{45}$ $-\mathbf{f}_{24}$	
2	5	$-\mathbf{f}_{05}-\mathbf{f}_{35}$	$-\mathbf{f}_{15}-\mathbf{f}_{45}$ $-\mathbf{f}_{25}$	

Note that if we compare this table with the table that shows the block partition, it's clear that the cyclic partition does a better job of balancing the load.

To implement this, during the first phase our revised algorithm proceeds as before, except that each thread adds the forces it computes into its own subarray of locforces:

```
# pragma omp for for each particle q {
        forceqk[X] = forceqk[Y] = 0 ; for each particle k > q {

            x diff = pos[q][X] − pos[k][X]; y diff = pos[q][Y] − pos[k][Y]; dist = sqrt(x diff
            x diff + y diff y diff);
    *         *
            distcubed = dist*dist*dist; forceqk[X] = G*masses[q]*masses[k]/distcubed *
            xdiff; forceqk[Y] = G*masses[q]*masses[k]/distcubed * y_diff;

                                locforces[myrank][q][X] +=
            forceqk[X];         locforces[myrank][q][Y] +=
            forceqk[Y];         locforces[myrank][k][X] −=
            forceqk[X];         locforces[myrank][k][Y] −=
            forceqk[Y];
        }
    }
```

During the second phase, each thread adds the forces computed by all the threads for its assigned particles:

```
# pragma omp for for (q = 0; q < n; q++) {
        forces[q][X] = forces[q][Y] = 0 ; for (thread = 0; thread < threadcount; thread++) {
        forces[q][X] += locforces[thread][q][X]; forces[q][Y] += loc_forces[thread][q][Y];


        }
    }
```

Before moving on, we should make sure that we haven't inadvertently introduced any new race conditions. During the first phase, since each thread writes to its own subarray, there isn't a race condition in the updates to locforces. Also, during the second phase, only the "owner" of thread *q* writes to forces[q], so there are no race conditions in the second phase. Finally, since there is an implied barrier after each of the parallelized **for** loops, we don't need to worry that some thread is going to race ahead and make use of a variable that hasn't been properly initialized, or that some slow thread is going to make use of a variable that has had its value changed by another thread.

Evaluating the OpenMP codes

Before we can compare the basic and the reduced codes, we need to decide how to schedule the parallelized **for** loops. For the basic code, we've seen that any schedule that divides the

iterations equally among the threads should do a good job of balancing the computational load. (As usual, we're assuming no more than one thread/core.) We also observed that a block partitioning of the iterations would result in fewer cache misses than a cyclic partition. Thus, we would expect that a block schedule would be the best option for the basic version.

In the reduced code, the amount of work done in the first phase of the computation of the forces decreases as the **for** loop proceeds. We've seen that a cyclic schedule should do a better job of assigning more or less equal amounts of work to each thread. In the remaining parallel **for** loops—the initialization of the locforces array, the second phase of the computation of the forces, and the updating of the positions and velocities—the work required is roughly the same for all the iterations. Therefore, *taken out of context* each of these loops will probably perform best with a block schedule. However, the schedule of one loop can affect the performance of another (see Exercise 6.10), so it may be that choosing a cyclic schedule for one loop and block schedules for the others will degrade performance.

With these choices, Table 6.3 shows the performance of the *n*-body solvers when they're run on one of our systems with no I/O. The solver used 400 particles for 1000 timesteps. The column labeled "Default Sched" gives times for the OpenMP reduced solver when all of the inner loops use the default schedule, which, on our system, is a block schedule. The column labeled "Forces Cyclic" gives times when the first phase of the forces computation uses a cyclic schedule and the other inner loops use the default schedule. The last column, labeled "All Cyclic," gives times when all of

Table 6.3 Run-Times of the *n*-Body Solvers Parallelized with OpenMP (times are in seconds)

Threads	Basic	Reduced Default Sched		Reduced Forces Cyclic		Reduced All Cyclic	
1	7.71	3.90	3.90	3.90	2	3.87	
2.94	1.98	2.01	4	1.95	1.73	1.01	
1.08							
8	0.99	0.95	0.54	0.61			

the inner loops use a cyclic schedule. The run-times of the serial solvers differ from those of the single-threaded solvers by less than 1%, so we've omitted them from the table.

Notice that with more than one thread the reduced solver, using all default schedules, takes anywhere from 50 to 75% longer than the reduced solver with the cyclic forces computation. Using the cyclic schedule is clearly superior to the default schedule in this case, and any loss in time resulting from cache issues is more than made up for by the improved load balance for the computations.

For only two threads there is very little difference between the performance of the reduced solver with only the first forces loop cyclic and the reduced solver with all loops cyclic. However, as we increase the number of threads, the performance of the reduced solver that uses a cyclic schedule for all of the loops does start to degrade. In this particular

case, when there are more threads, it appears that the overhead involved in changing distributions is less than the overhead incurred from false sharing.

Finally, notice that the basic solver takes about twice as long as the reduced solver with the cyclic scheduling of the forces computation. So if the extra memory is available, the reduced solver is clearly superior. However, the reduced solver increases the memory requirement for the storage of the forces by a factor of threadcount, so for very large numbers of particles, it may be impossible to use the reduced solver.

Parallelizing the solvers using pthreads

Parallelizing the two *n*-body solvers using Pthreads is very similar to parallelizing them using OpenMP. The differences are only in implementation details, so rather than repeating the discussion, we will point out some of the principal differences between the Pthreads and the OpenMP implementations. We will also note some of the more important similarities.

- By default local variables in Pthreads are private, so all shared variables are global in the Pthreads version.

The principal data structures in the Pthreads version are identical to those in the OpenMP version: vectors are two-dimensional arrays of **double**s, and the mass, position, and velocity of a single particle are stored in a struct. The forces are stored in an array of vectors.

Startup for Pthreads is basically the same as the startup for OpenMP: the main thread gets the command-line arguments, and allocates and initializes the principal data structures.

The main difference between the Pthreads and the OpenMP implementations is in the details of parallelizing the inner loops. Since Pthreads has nothing analogous to a parallel **for** directive, we must explicitly determine which values of the loop variables correspond to each thread's calculations. To facilitate this, we've

- written a function Loopschedule, which determines
- the initial value of the loop variable,
- the final value of the loop variable, and the increment for the loop variable.
- The input to the function is
- the calling thread's rank,

• the number of threads, • the total number of iterations, and an argument indicating whether the partitioning should be block or cyclic. Another difference between the Pthreads and the OpenMP versions has to do with barriers. Recall that the end of a parallel **for** directive in OpenMP has an implied barrier. As we've seen, this is important. For example, we don't want a thread to start updating its positions until all the forces have been calculated, because it could use an out-of-date force and another thread could use an outof-date position. If we simply partition the loop iterations among the threads in the Pthreads version, there won't be a barrier at the end of an inner **for** loop and we'll have a race condition. Thus, we need to add explicit barriers after the inner loops when a race condition can arise. The Pthreads standard includes a barrier. However, some systems don't implement it, so we've defined a function that uses a Pthreads condition variable to implement a barrier. See Subsection 4.8.3 for details.

Parallelizing the basic solver using MPI

With our composite tasks corresponding to the individual particles, it's fairly straightforward to parallelize the basic algorithm using MPI. The only communication among the tasks occurs when we're computing the forces, and, in order to compute the forces, each task/particle needs the position and mass of every other particle. MPIAllgather is expressly designed for this situation, since it collects on each process the same information from every other process. We've already noted that a block distribution will probably have the best performance, so we should use a block mapping of the particles to the processes.

In the shared-memory implementations, we collected most of the data associated with a single particle (mass, position, and velocity) into a single struct. However, if we use this data structure in the MPI implementation, we'll need to use a derived datatype in the call to MPIAllgather, and communications with derived datatypes tend to be slower than communications with basic MPI types. Thus, it will make more sense to use individual arrays for the masses, positions, and velocities. We'll also need an array for storing the positions of all the particles. If each process has sufficient memory, then each of these can be a separate array. In fact, if memory isn't a problem, each process can store the entire array of masses, since these will never be updated and their values only need to be communicated during the initial setup.

On the other hand, if memory is short, there is an "in-place" option that can be used with some MPI collective communications. For our situation, suppose that the array pos can store the positions of all n particles. Further suppose that vectmpit is an MPI datatype that stores two contiguous **double**s. Also suppose that n is evenly divisible by commsz and locn = n/commsz. Then, if we store the local positions in a separate array, locpos, we can use the following call to collect all of the positions

on each process:

```
MPIAllgather(locpos, locn, vectmpit, pos, locn, vectmpit, comm);
```

If we can't afford the extra storage for locpos, then we can have each process q store its local positions in the qth block of pos. That is, the local positions of each process should be stored in the appropriate block of each process' pos array:

```
Process 0: pos[0], pos[1], . . . , pos[locn−1]

Process 1: pos[locn], pos[locn+1], . . . , pos[locn + locn−1]

. . .
Process q: pos[q∗locn], pos[q∗locn+1], . . . , pos[q∗locn + locn−1]

. . .
```

With the pos array initialized this way on each process, we can use the following call to MPIAllgather:

```
MPIAllgather(MPI_IN_PLACE, locn, vectmpit, pos, locn, vectmpit, comm);
```

In this call, the first locn and vectmpit arguments are ignored. However, it's not a bad idea to use arguments whose values correspond to the values that will be used, just to increase the readability of the program.

In the program we've written, we made the following choices with respect to the data structures:

- Each process stores the entire global array of particle masses.
- Each process only uses a single n-element array for the positions.
- Each process uses a pointer locpos that refers to the start of its block of pos.

Thus, on process, 0 localpos = pos, on process 1 localpos = pos + locn, and, so on.

With these choices, we can implement the basic algorithm with the pseudocode shown in Program 6.2. Process 0 will read and broadcast the command line arguments. It will also read the input and print the results. In Line 1, it will need to distribute the input data. Therefore, Get input data might be implemented as follows:

```
if (my rank == 0) {
      for each particle
Read masses[particle], pos[particle], vel[particle];
   }
      MPIBcast(masses, n, MPIDOUBLE, 0, comm);
      MPIBcast(pos, n, vect mpi t, 0, comm);
      MPIScatter(vel, loc n, vect mpi t, locvel, loc n, vect mpi t, 0 , comm);
```

So process 0 reads all the initial conditions into three n-element arrays. Since we're storing all the masses on each process, we broadcast masses. Also, since each process

```
1    Get input data;
2    for each timestep {
3    if (timestep output)
4    Print positions and velocities of particles;
5    for each local particle loc q
6    Compute total force on loc q;
7    for each local particle loc q
8    Compute position and velocity of loc q;
9    Allgather local positions into global pos array;
10   }
11   Print positions and velocities of particles;
```

Program 6.2: Pseudocode for the MPI version of the basic *n*-body solver

will need the global array of positions for the first computation of forces in the main **for** loop, we just broadcast pos. However, velocities are only used locally for the updates to positions and velocities, so we scatter vel.

Notice that we gather the updated positions in Line 9 at the end of the body of the outer **for** loop of Program 6.2. This insures that the positions will be available for output in both Line 4 and Line 11. If we're printing the results for each timestep, this placement allows us to eliminate an expensive collective communication call: if we simply gathered the positions onto process 0 before output, we'd have to call MPIAllgather before the computation of the forces. With this organization of the body of the outer **for** loop, we can implement the output with the following pseudocode:

```
Gather velocities onto process 0 ; if (my rank == 0) {
        –
    Print timestep; for each
    particle
        Print pos[particle] and vel[particle]
}
```

Parallelizing the reduced solver using MPI

The "obvious" implementation of the reduced algorithm is likely to be extremely complicated. Before computing the forces, each process will need to gather a subset of the positions, and after the computation of the forces, each process will need to scatter some of the individual forces it has computed and add the forces it receives. Figure 6.6 shows the communications that would take place if we had three processes, six particles, and used a block partitioning of the particles among the processes. Not suprisingly, the communications are even more complex when we use a cyclic distribution (see Exercise 6.13). Certainly it would be possible to implement these communications. However, unless the implementation were *very* carefully done, it would probably be *very* slow.

Fortunately, there's a much simpler alternative that uses a communication

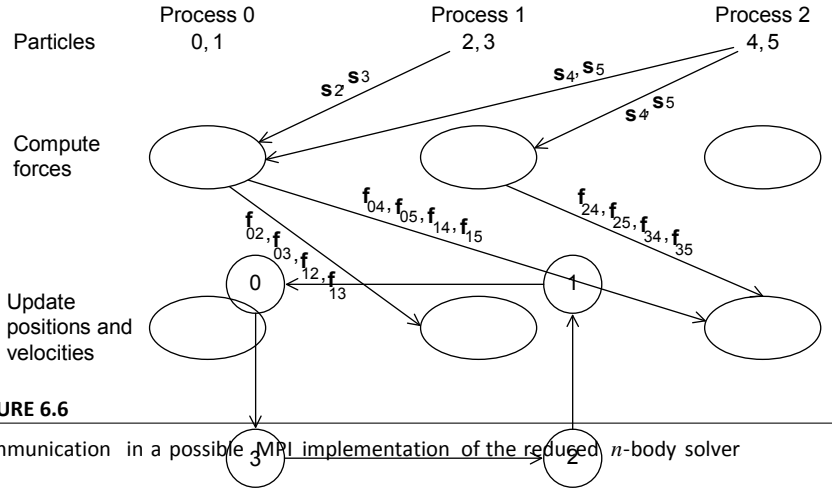


FIGURE 6.6

Communication in a possible MPI implementation of the reduced n-body solver

FIGURE 6.7

A ring of processes

structure that is sometimes called a **ring pass**. In a ring pass, we imagine the processes as being interconnected in a ring (see Figure 6.7). Process 0 communicates directly with processes 1 and commsz−1, process 1 communicates with processes 0 and 2 , and so on. The communication in a ring pass takes place in phases, and during each phase each process sends data to its "lower-ranked" neighbor, and receives data from its "higher-ranked" neighbor. Thus, 0 will send to commsz−1 and receive from 1. 1 will send to 0 and receive from 2, and so on. In general, process q will send to process $(q-1+\text{commsz})\%\text{commsz}$ and receive from process $(q+1)\%\text{commsz}$.

By repeatedly sending and receiving data using this ring structure, we can arrange that each process has access to the positions of all the particles. During the first phase, each process will send the positions of its assigned particles to its "lower-ranked" neighbor and receive the positions of the particles assigned to its higher-ranked neighbor. During the next phase, each process will forward the positions it received in the first phase. This process continues through commsz−1

phases until each process has received the positions of all of the particles. Figure 6.8 shows the three phases if there are four processes and eight particles that have been cyclically distributed.

Of course, the virtue of the reduced algorithm is that we don't need to compute all of the inter-particle forces since $\mathbf{f}_{kq} = -\mathbf{f}_{qk}$, for every pair of particles q and k. To see

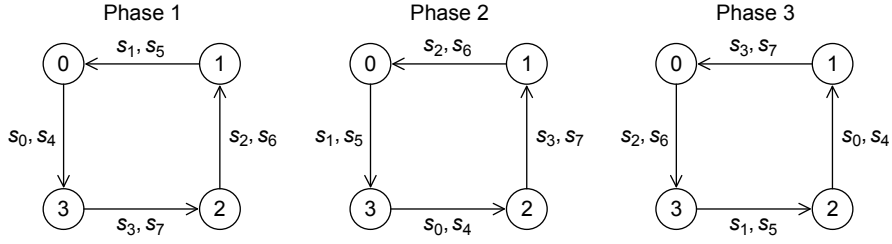


FIGURE 6.8

Ring pass of positions

how to exploit this, first observe that using the reduced algorithm, the interparticle forces can be divided into those that are *added* into and those that are subtracted from the total forces on the particle. For example, if we have six particles, then the reduced algorithm will compute the force on particle 3 as

$$\mathbf{F}_3 = -\mathbf{f}_{03} - \mathbf{f}_{13} - \mathbf{f}_{23} + \mathbf{f}_{34} + \mathbf{f}_{35}.$$

The key to understanding the ring pass computation of the forces is to observe that the interparticle forces that are *subtracted* are computed by another task/particle, while the forces that are *added* are computed by the owning task/particle. Thus, the computations of the interparticle forces on particle 3 are assigned as follows:

Force	$\mathbf{f}_{03}$	$\mathbf{f}_{13}$	$\mathbf{f}_{23}$	$\mathbf{f}_{34}$	$\mathbf{f}_{35}$
Task/Particle	0	1	2	3	3

So, suppose that for our ring pass, instead of simply passing locn = n/commsz positions, we also pass locn forces. Then in each phase, a process can

1. compute interparticle forces resulting from interaction between its assigned particles and the particles whose positions it has received, and

2. once an interparticle force has been computed, the process can add the force into a local array of forces corresponding to its particles, *and* it can subtract the interparticle force from the received array of forces.

See, for example, [15, 34] for further details and alternatives.

Let's take a look at how the computation would proceed when we have four particles, two processes, and we're using a cyclic distribution of the particles among the processes (see Table 6.4). We're calling the arrays that store the local positions and local forces locpos and locforces, respectively. These are not communicated among the processes. The arrays that are communicated among the processes are tmppos and tmpforces.

Before the ring pass can begin, both arrays storing positions are initialized with the positions of the local particles, and the arrays storing the forces are set to 0. Before the ring pass begins, each process computes those forces that are due to interaction

Table 6.4 Computation of Forces in Ring Pass

Time	Variable	Process 0	Process 1

Start	locpos $\mathbf{s}_0,\mathbf{s}_2$	$\mathbf{s}_1,\mathbf{s}_3$	locforces 0,0 0 , 0 tmppos $\mathbf{s}_0,\mathbf{s}_2$ $\mathbf{s}_1,\mathbf{s}_3$ tmpforces 0,0 0 , 0
After Comp of Forces	locpos $\mathbf{s}_0,\mathbf{s}_2$ locforces $\mathbf{f}_{02},0$ tmppos $\mathbf{s}_0,\mathbf{s}_2$	$\mathbf{s}_1,\mathbf{s}_3$ $\mathbf{f}_{13},0$ $\mathbf{s}_1,\mathbf{s}_3$	tmpforces $0,-\mathbf{f}_{02}$ $0,-\mathbf{f}_{13}$
After First Comm	locpos $\mathbf{s}_0,\mathbf{s}_2$ locforces $\mathbf{f}_{02},0$ tmppos $\mathbf{s}_1,\mathbf{s}_3$	$\mathbf{s}_1,\mathbf{s}_3$ $\mathbf{f}_{13},0$ $\mathbf{s}_0,\mathbf{s}_2$	tmpforces $0,-\mathbf{f}_{13}$ $0,-\mathbf{f}_{02}$
After Comp of Forces	locpos $\mathbf{s}_0,\mathbf{s}_2$ locforces $\mathbf{f}_{01}+\mathbf{f}_{02}+\mathbf{f}_{03},\mathbf{f}_{23}$ tmppos $\mathbf{s}_1,\mathbf{s}_3$	$\mathbf{s}_1,\mathbf{s}_3$ $\mathbf{f}_{12}+\mathbf{f}_{13},0$ $\mathbf{s}_0,\mathbf{s}_2$	tmpforces $-\mathbf{f}_{01},-\mathbf{f}_{03}-\mathbf{f}_{13}-\mathbf{f}_{23}$ $0,-\mathbf{f}_{02}-\mathbf{f}_{12}$
After Second Comm	locpos $\mathbf{s}_0,\mathbf{s}_2$ locforces $\mathbf{f}_{01}+\mathbf{f}_{02}+\mathbf{f}_{03},\mathbf{f}_{23}$ tmppos $\mathbf{s}_0,\mathbf{s}_2$	$\mathbf{s}_1,\mathbf{s}_3$ $\mathbf{f}_{12}+\mathbf{f}_{13},0$ $\mathbf{s}_1,\mathbf{s}_3$	tmpforces $0,-\mathbf{f}_{02}-\mathbf{f}_{12}$ $-\mathbf{f}_{01},-\mathbf{f}_{03}-\mathbf{f}_{13}-\mathbf{f}_{23}$
After Comp of Forces	locpos $\mathbf{s}_0,\mathbf{s}_2$ locforces $\mathbf{f}_{01}+\mathbf{f}_{02}+\mathbf{f}_{03},-\mathbf{f}_{02}-\mathbf{f}_{12}+\mathbf{f}_{23}$ tmppos $\mathbf{s}_0,\mathbf{s}_2$	$\mathbf{s}_1,\mathbf{s}_3$ $-\mathbf{f}_{01}+\mathbf{f}_{12}+\mathbf{f}_{13},-\mathbf{f}_{03}-\mathbf{f}_{13}-\mathbf{f}_{23}$ $\mathbf{s}_1,\mathbf{s}_3$	tmpforces $0,-\mathbf{f}_{02}-\mathbf{f}_{12}$ $-\mathbf{f}_{01},-\mathbf{f}_{03}-\mathbf{f}_{13}-\mathbf{f}_{23}$

among its assigned particles. Process 0 computes $\mathbf{f}_{02}$ and process 1 computes $\mathbf{f}_{13}$. These values are added into the appropriate locations in locforces and subtracted from the appropriate locations in tmpforces.

Now, the two processes exchange tmppos and tmpforces and compute the forces due to interaction among their local particles and the received particles. In the reduced algorithm, the lower ranked task/particle carries out the computation. Process 0 computes $\mathbf{f}_{01}$,$\mathbf{f}_{03}$, and $\mathbf{f}_{23}$, while process 1 computes $\mathbf{f}_{12}$. As before, the newly computed forces are added into the appropriate locations in locforces and subtracted from the appropriate locations in tmpforces.

To complete the algorithm, we need to exchange the tmp arrays one final time.[1] Once each process has received the updated tmpforces, it can carry out a simple vector sum locforces += tmpforces to complete the algorithm.

```
1  source = ( my_rank + 1) % comm_sz;
2  dest = ( my_rank − 1 + comm_sz) % comm_sz;
3  Copy loc_pos into tmp_pos;
4  loc_forces = tmp_forces = 0;
5
6  Compute forces due to interactions among local particles;
7  for ( phase = 1; phase < comm_sz; phase++) {
8      Send current tmp_pos and tmp_forces to dest;
9      Receive new tmp_pos and tmp_forces from source;
10     /* Owner of the positions and forces we're receiving */
11     owner = ( my_rank + phase) % comm_sz;
12     Compute forces due to interactions among my particles
13         and owner's particles;
14 }
15 Send current tmp_pos and tmp_forces to dest;
16 Receive new tmp_pos and tmp_forces from source;
```

Program 6.3: Pseudocode for the MPI implementation of the reduced *n*-body solver

Thus, we can implement the computation of the forces in the reduced algorithm using a ring pass with the pseudocode shown in Program 6.3. Recall that using MPISend and MPIRecv for the send-receive pairs in Lines 8 and 9 and 15 and 16 is *unsafe* in MPI parlance, since they can hang if the system doesn't provide sufficient buffering. In this setting, recall that MPI provides MPISendrecv and MPISendrecvreplace. Since we're using the same memory for both the outgoing and the incoming data, we can use MPISendrecvreplace.

Also recall that the time it takes to start up a message is substantial. We can probably reduce the cost of the communication by using a single array to store both

[1] Actually, we only need to exchange tmpforces for the final communication.

tmppos and tmpforces. For example, we could allocate storage for an array tmpdata that can store 2×locn objects with type vectt and use the first locn for tmppos and the last locn for tmpforces. We can continue to use tmppos and tmpforces by making these pointers to tmpdata[0] and tmpdata[locn], respectively.

The principal difficulty in implementing the actual computation of the forces in Lines 12 and 13 lies in determining whether the current process should compute the force resulting from the interaction of a particle q assigned to it and a particle r whose position it has received. If we recall the reduced algorithm (Program 6.1), we see that task/particle q is responsible for computing $\mathbf{f}_{qr}$ if and only if $q < r$. However, the arrays locpos and tmppos (or a larger array containing tmppos and tmpforces) use *local* subscripts, not global subscripts. That is, when we access an element of (say) locpos, the subscript we use will lie in the range 0,1,...,locn−1, not 0,1,...,n−1; so, if we try to implement the force interaction with the following pseudocode, we'll run into (at least) a couple of problems:

```
for (locpart1 = 0; locpart1 < loc n−1; locpart1++)
    for (locpart2 = loc_part1+1; locpart2 < loc n; locpart2++)

        Computeforce(locpos[locpart1], masses[locpart1], tmp_pos[locpart2],
                masses[locpart2], locforces[locpart1], tmpforces[locpart2]);
```

The first, and most obvious, is that masses is a global array and we're using local subscripts to access its elements. The second is that the relative sizes of locpart1 and locpart2 don't tell us whether we should compute the force due to their interaction. We need to use global subscripts to determine this. For example, if we have four particles and two processes, and the preceding code is being run by process 0 , then when locpart1 = 0, the inner loop will skip locpart2 = 0 and start with locpart2 = 1; however, if we're using a cyclic distribution, locpart1 = 0 corresponds to global particle 0 and locpart2 = 0 corresponds to global particle 1, and we *should* compute the force resulting from interaction between these two particles.

Clearly, the problem is that we shouldn't be using local particle indexes, but rather we should be using *global* particle indexes. Thus, using a cyclic distribution of the particles, we could modify our code so that the loops also iterate through global particle indexes:

```
for (locpart1 = 0, glbpart1 = my rank; locpart1 < loc n−1 ;
        locpart1++, glbpart1 += comm sz)
    for (glbpart2 = Firstindex(glbpart1, myrank, owner, comm sz), locpart2 = Global to
        local(glbpart2, owner, loc n); locpart2 < loc n;

        locpart2++, glbpart2 += comm sz)
        Computeforce(locpos[locpart1], masses[glbpart1], tmp pos[locpart2],
            masses[glbpart2], locforces[locpart1], tmpforces[locpart2]);
```

The function Firstindex should determine a global index glbpart2 with the following properties:

1. The particle glbpart2 is assigned to the process with rank owner.
2. glbpart1<glbpart2<glbpart1 + commsz.

The function Globaltolocal should convert a global particle index into a local particle index, and the function Computeforce should compute the force resulting from the interaction of two particles. We already know how to implement Computeforce. See Exercises 6.15 and 6.16 for the other two functions.

Performance of the MPI solvers

Table 6.5 shows the run-times of the two *n*-body solvers when they're run with 800 particles for 1000 timesteps on an Infiniband-connected cluster. All the timings were taken with one process per cluster node. The run-times of the serial solvers differed from the single-process MPI solvers by less than 1%, so we haven't included them.

Clearly, the performance of the reduced solver is much superior to the performance of the basic solver, although the basic solver achieves higher efficiencies.

Table 6.5 Performance of the MPI *n*-Body Solvers (times in seconds)

Processes	Basic	Reduced
1	17.30	8.68
2	8.65	4.45
4	4.35	2.30
8	2.20	1.26
16	1.13	0.78

Table 6.6 Run-Times for OpenMP and MPI *n*- Body Solvers (times in seconds)

Processes/	OpenMP		MPI	
Threads	Basic	Reduced	Basic	Reduced
1	15.13	8.77	17.30	8.68
2	7.62	4.42	8.65	4.45
4	3.85	2.26	4.35	2.30

For example, the efficiency of the basic solver on 16 nodes is about 0.95, while the efficiency of the reduced solver on 16 nodes is only about 0.70.

A point to stress here is that the reduced MPI solver makes much more efficient use of memory than the basic MPI solver; the basic solver must provide storage for all *n* positions on each process, while the reduced solver only needs extra storage for n/commsz positions and n/commsz forces. Thus, the extra storage needed on each process for the basic solver is nearly $\text{commsz}/2$ times greater than the storage needed for the reduced solver. When *n* and commsz are very large, this factor can easily make the difference between being able to run a simulation only using the process' main memory and having to use secondary storage.

The nodes of the cluster on which we took the timings have four cores, so we can compare the performance of the OpenMP implementations with the performance of the MPI implementations (see Table 6.6). We see that the basic OpenMP solver is a good deal faster than the basic MPI solver. This isn't surprising since MPIAllgather is such an expensive operation. Perhaps surprisingly, though, the reduced MPI solver is quite competitive with the reduced OpenMP solver.

Let's take a brief look at the amount of memory required by the MPI and OpenMP reduced solvers. Say that there are n particles and p threads or processes. Then each solver will allocate the same amount of storage for the local velocities and the local positions. The MPI solver allocates n **double**s per process for the masses. It also allocates $4n/p$ **double**s for the tmppos and tmpforces arrays, so in addition to the

local velocities and positions, the MPI solver stores

$n+4n/p$

doubles per process. The OpenMP solver allocates a total of $2pn+2n$ **double**s for the forces and n **double**s for the masses, so in addition to the local velocities and positions, the OpenMP solver stores

$3n/p+2n$

doubles per thread. Thus, the difference in the local storage required for the OpenMP version and the MPI version is

$n-n/p$

doubles. In other words, if n is large, the local storage required for the MPI version is substantially less than the local storage required for the OpenMP version. So, for a fixed number of processes or threads, we should be able to run much larger simulations with the MPI version than the OpenMP version. Of course, because of hardware considerations, we're likely to be able to use many more MPI processes than OpenMP threads, so the size of the largest possible MPI simulations should be *much* greater than the size of the largest possible OpenMP simulations. The MPI version of the reduced solver is much more scalable than any of the other versions, and the "ring pass" algorithm provides a genuine breakthrough in the design of n-body solvers.

TREE SEARCH

Many problems can be solved using a tree search. As a simple example, consider the traveling salesperson problem, or TSP. In TSP, a salesperson is given a list of cities she needs to visit and a cost for traveling between each pair of cities. Her problem is to visit each city once, returning to her hometown, and she must do this with the least possible cost. A route that starts in her hometown, visits each city once and returns to her hometown is called a *tour*; thus, the TSP is to find a minimum-cost tour.

Unfortunately, TSP is what's known as an **NP-complete** problem. From a practical standpoint, this means that there is no algorithm known for solving it that, in all cases, is significantly better than exhaustive search. Exhaustive search means examining all possible solutions to the problem and choosing the best. The number of possible solutions to TSP grows exponentially as the number of cities is increased. For example, if we add one additional city to an n-city problem, we'll increase the number of possible solutions by a factor of $n-1$. Thus, although there are only six possible solutions to a four-city problem, there are $4\times6 = 24$ to a five-city problem, $5\times24 = 120$ to a six-city problem, $6\times120 = 720$ to a seven-city

problem, and so on. In fact, a 100-city problem has far more possible solutions than the number of atoms in the universe!

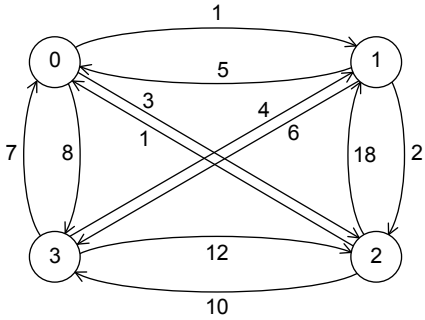


FIGURE 6.9

A four-city TSP

Furthermore, if we could find a solution to TSP that's significantly better in all cases than exhaustive search, then there are literally hundreds of other very hard problems for which we could find fast solutions. Not only is there no known solution to TSP that is better in all cases than exhaustive search, it's very unlikely that we'll find one.

So how can we solve TSP? There are a number of clever solutions. However, let's take a look at an especially simple one. It's a very simple form of tree search. The idea is that in searching for solutions, we build a *tree.* The leaves of the tree correspond to tours, and the other tree nodes correspond to "partial" tours—routes that have visited some, but not all, of the cities.

Each node of the tree has an associated cost, that is, the cost of the partial tour. We can use this to eliminate some nodes of the tree. Thus, we want to keep track of the cost of the best tour so far, and, if we find a partial tour or node of the tree that couldn't possibly lead to a less expensive complete tour, we shouldn't bother searching the children of that node (see Figures 6.9 and 6.10).

In Figure 6.9 we've represented a four-city TSP as a labeled, directed graph. A **graph** (not to be confused with a graph in calculus) is a collection of vertices and edges or line segments joining pairs of vertices. In a **directed graph** or **digraph**, the edges are oriented—one end of each edge is the tail, and the other is the head. A graph or digraph is **labeled** if the vertices and/or edges have labels. In our example, the vertices of the digraph correspond to the cities in an instance of the TSP, the edges correspond to routes between the cities, and the labels on the edges correspond to the costs of the routes. For example, there's a cost of 1 to go from city 0 to city 1 and a cost of 5 to go from city 1 to city 0.

If we choose vertex 0 as the salesperson's home city, then the initial partial tour consists only of vertex 0, and since we've gone nowhere, it's cost is 0. Thus, the root of the tree in Figure 6.10 has the partial tour consisting only of the vertex 0

with cost 0. From 0 we can first visit 1, 2, or 3, giving us three two-city partial tours with costs 1, 3, and 8, respectively. In Figure 6.10 this gives us three children of the root. Continuing, we get six three-city partial tours, and since there are

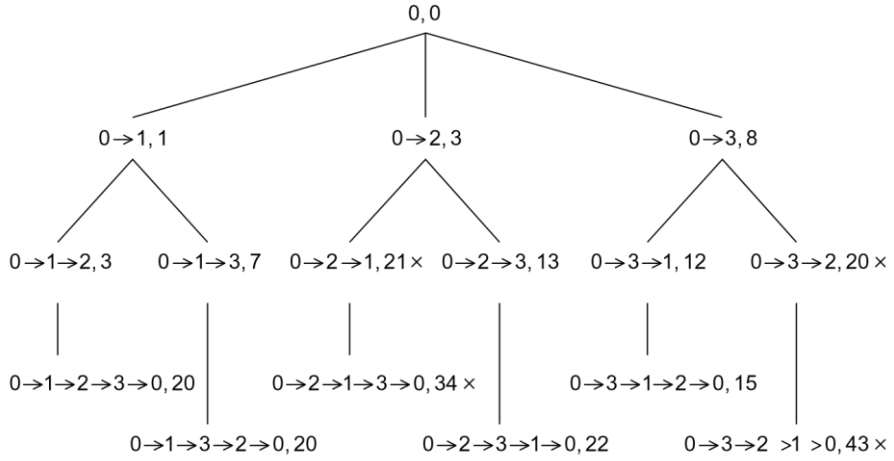


FIGURE 6.10

Search tree for four-city TSP

only four cities, once we've chosen three of the cities, we know what the complete tour is.

Now, to find a least-cost tour, we should search the tree. There are many ways to do this, but one of the most commonly used is called **depth-first search**. In depthfirst search, we probe as deeply as we can into the tree. After we've either reached a leaf or found a tree node that can't possibly lead to a least-cost tour, we back up to the deepest "ancestor" tree node with unvisited children, and probe one of its children as deeply as possible.

In our example, we'll start at the root, and branch left until we reach the leaf labeled

0 → 1 → 2 → 3 → 0, Cost 20.

Then we back up to the tree node labeled 0 → 1, since it is the deepest ancestor node with unvisited children, and we'll branch down to get to the leaf labeled

0 → 1 → 3 → 2 → 0, Cost 20.

Continuing, we'll back up to the root and branch down to the node labeled 0 → 2. When we visit its child, labeled

0 → 2 → 1, Cost 21 ,

we'll go no further in this subtree, since we've already found a complete tour with cost less than 21. We'll back up to 0 → 2 and branch down to its remaining unvisited child. Continuing in this fashion, we eventually find the least-cost tour

0 → 3 → 1 → 2 → 0, Cost 15.

Recursive depth-first search

Using depth-first search we can systematically visit each node of the tree that could possibly lead to a least-cost solution. The simplest formulation of depth-first search uses recursion (see Program 6.4). Later on it will be useful to have a definite order in which the cities are visited in the **for** loop in Lines 8 to 13, so we'll assume that the cities are visited in order of increasing index, from city 1 to city $n-1$. The algorithm makes use of several global variables:

- n: the total number of cities in the problem digraph: a data structure representing the input digraph hometown: a data structure representing vertex or city 0, the
- salesperson's hometown
- besttour: a data structure representing the best tour so far

The function Citycount examines the partial tour tour to see if there are n cities on the partial tour. If there are, we know that we simply need to return to the hometown to complete the tour, and we can check to see if the complete tour has a lower cost than the current "best tour" by calling Besttour. If it does, we can replace the current best tour with this tour by calling the function Updatebesttour. Note that before the first call to Depthfirstsearch, the besttour variable should be initialized so that its cost is greater than the cost of any possible least-cost tour.

If the partial tour tour hasn't visited n cities, we can continue branching down in the tree by "expanding the current node," in other words, by trying to visit other

```
1  void Depth_first_search(tour_t tour) {
2      city_t city;
3
4      if (City_count(tour) == n) {
5          if (Best_tour(tour))
6              Update_best_tour(tour);
7      } else {
8          for each neighboring city
9              if (Feasible(tour, city)) {
10                 Add_city(tour, city);
11                 Depth_first_search(tour);
12                 Remove_last_city(tour, city);
13             }
14     }
15 } /* Depth_first_search */
```

cities from the city last visited in the partial tour. To do this we simply loop through the cities. The function Feasible checks to see if the city or vertex has already

Program 6.4: Pseudocode for a recursive solution to TSP using depth-first search

we return from Depthfirstsearch, we remove the city from the tour, since it shouldn't be included in the tour used in subsequent recursive calls.

Nonrecursive depth-first search

Since function calls are expensive, recursion can be slow. It also has the disadvantage that at any given instant of time only the current tree node is accessible. This could be a problem when we try to parallelize tree search by dividing tree nodes among the threads or processes.

It is possible to write a nonrecursive depth-first search. The basic idea is modeled on recursive implementation. Recall that recursive function calls can be implemented by pushing the current state of the recursive function onto the run-time stack. Thus, we can try to eliminate recursion by pushing necessary data on our own stack before branching deeper into the tree, and when we need to go back up the tree—either because we've reached a leaf or because we've found a node that can't lead to a better solution—we can pop the stack.

This outline leads to the implementation of iterative depth-first search shown in Program 6.5. In this version, a stack record consists of a single city, the city that will be added to the tour when its record is popped. In the recursive version we continue to make recursive calls until we've visited every node of the tree that corresponds to a feasible partial tour. At this point, the stack won't have any more activation records

for calls to Depth-first-search, and we'll return to the function that made the

```
1  for ( city = n-1; city >= 1; city -- )
2      Push(stack, city);
3  while (! Empty(stack )) {
4      city = Pop(stack);
5      if ( city == NO-CITY) // End of child list, back up
6          Remove-last-city(curr tour);
7      else {
8          Add-city(curr-tour, city);
9          if ( City-count(curr-tour) == n) {
10             if ( Best-tour(curr-tour))
11                 Update-best-tour(curr-tour);
12             Remove-last-city(curr-tour);
13         } else {
14             Push(stack, NO-CITY);
15             for ( nbr = n-1; nbr >= 1; nbr -- )
16                 if ( Feasible(curr tour, nbr))
17                     Push(stack, nbr);
18         }
19     } /* if Feasible */
20 } /* while ! Empty */
```

Program 6.5: Pseudocode for an implementation of a depth-first solution to TSP that doesn't use recursion

original call to Depthfirstsearch. The main control structure in our iterative version is the **while** loop extending from Line 3 to Line 20, and the loop termination condition is that our stack is empty. As long as the search needs to continue, we need to make sure the stack is nonempty, and, in the first two lines, we add each of the non-hometown cities. Note that this loop visits the cities in decreasing order, from $n-1$ down to 1. This is because of the order created by the stack, whereby the stack pops

the top cities first. By reversing the order, we can insure that the cities are visited in the same order as the recursive function.

Also notice that in Line 5 we check whether the city we've popped is the constant NOCITY. This constant is used so that we can tell when we've visited all of the children of a tree node; if we didn't use it, we wouldn't be able to tell when to back up in the tree. Thus, before pushing all of the children of a node (Lines 15–17), we push the NOCITY marker.

An alternative to this iterative version uses partial tours as stack records (see Program 6.6). This gives code that is closer to the recursive function. However, it also results in a slower version, since it's necessary for the function that pushes onto the stack to create a copy of the tour before actually pushing it on to the stack. To emphasize this point, we've called the function Pushcopy. (What happens if we simply push a pointer to the current tour onto the stack?) The extra memory required will probably not be a problem. However, allocating storage for a new tour and copying the existing tour is time-consuming. To some degree we can mitigate these costs by saving freed tours in our own data structure, and when a freed tour is available we can use it in the Pushcopy function instead of calling malloc.

On the other hand, this version has the virtue that the stack is more or less independent of the other data structures. Since entire tours are stored, multiple threads or processes can "help themselves" to tours, and, if this is done reasonably carefully,

```
3       curr_tour = Pop(stack);
4       if ( City_count(curr_tour) == n) {
5           if ( Best_tour(curr_tour))
6               Update_best_tour(curr_tour);
7       } else {
8           for ( nbr = n-1; nbr >= 1; nbr-- )
9               if ( Feasible(curr_tour, nbr)) {
10                  Add_city(curr_tour, nbr);
11                  Push_copy(stack, curr_tour);
12                  Remove_last_city(curr_tour);
13              }
14      }
15      Free_tour(curr_tour);
16  }
```

Program 6.6: Pseudocode for a second solution to TSP that doesn't use recursion

it won't destroy the correctness of the program. With the original iterative version, a stack record is just a city and it doesn't provide enough information by itself to show where we are in the tree.

Data structures for the serial implementations

Our principal data structures are the tour, the digraph, and, in the iterative implementations, the stack. The tour and the stack are essentially list structures. In problems that we're likely to be able to tackle, the number of cities is going to be small— certainly less than 100—so there's no great advantage to using a linked list to represent the tours and we've used an array that can store n+1 cities. We repeatedly need both the number of cities in the partial tour and the cost of the partial tour. Therefore, rather than just using an array for the tour data structure and recomputing these values, we use a struct with three members: the array storing the cities, the number of cities, and the cost of the partial tour.

To improve the readability and the performance of the code, we can use preprocessor macros to access the members of the struct. However, since macros can be a nightmare to debug, it's a good idea to write "accessor" functions for use during initial development. When the program with accessor functions is working, they can be replaced with macros. As an example, we might start with the function

```
/* Find the ith city on the partial tour */ int Tourcity(tour_t
tour, int i) { return tour-> cities[i];

}/* Tourcity */
```

When the program is working, we could replace this with the macro

```
/* Find the ith city on the partial tour */
#define Tourcity(tour, i) (tour-> cities[i ])
```

The stack in the original iterative version is just a list of cities or **int**s. Furthermore, since there can't be more than $n^2/2$ records on the stack (see Exercise 6.17) at any one time, and n is likely to be small, we can just use an array, and like

the tour data structure, we can store the number of elements on the stack. Thus, for example, Push can be implemented with

```
void Push(my_stack_t stack, int city) {
   int loc = stack->list.sz;
   stack->list[loc] = city; stack->list.sz++;
}  /* Push */
```

In the second iterative version, the version that stores entire tours in the stack, we can probably still use an array to store the tours on the stack. Now the push function will look something like this:

```
void Push_copy(my_stack_t stack, tour_t tour) {
   int loc = stack->list.sz;
   tour_t tmp = Alloc_tour(); Copy_tour(tour, tmp); stack->list[loc] = tmp;
   stack->list.sz++;
}  /* Push */
```

Once again, element access for the stack can be implemented with macros.

There are many possible representations for digraphs. When the digraph has relatively few edges, list representations are preferred. However, in our setting, if vertex i is different from vertex j, there are directed, weighted edges from i to j and from j to i, so we need to store a weight for each ordered pair of distinct vertices i and j. Thus, in our setting, an **adjacency matrix** is almost certainly preferable to a list structure. This is an $n \times n$ matrix, in which the weight of the edge from vertex i to vertex j can be the entry in the ith row and jth column of the matrix. We can access this weight directly, without having to traverse a list. The diagonal elements (row i and column i) aren't used, and we'll set them to 0.

Performance of the serial implementations

The run-times of the three serial implementations are shown in Table 6.7. The input digraph contained 15 vertices (including the hometown), and all three algorithms visited approximately 95,000,000 tree nodes. The first iterative version is less than 5 % faster than the recursive version, and the second iterative version is about 8% slower than the recursive version. As expected, the first iterative solution eliminates

some of the overhead due to repeated function calls, while the second iterative solution is slower because of the repeated copying of tour data structures. However, as we'll see, the second iterative solution is relatively easy to parallelize, so we'll be using it as the basis for the parallel versions of tree search.

Parallelizing tree search

Let's take a look at parallelizing tree search. The tree structure suggests that we identify tasks with tree nodes. If we do this, the tasks will communicate down the tree edges: a parent will communicate a new partial tour to a child, but a child, except for terminating, doesn't communicate directly with a parent.

We also need to take into consideration the updating and use of the best tour. Each task examines the best tour to determine whether the current partial tour is feasible or the current complete tour has lower cost. If a leaf task determines its tour is a better tour, then it will also update the best tour. Although all of the actual computation can

Table 6.7 Run-Times of the Three Serial Implementations of Tree Search (times in seconds)

Recursive	First Iterative	Second Iterative
30.5	29.2	32.9

be considered to be carried out by the tree node tasks, we need to keep in mind that the best tour data structure requires additional communication that is not explicit in the tree edges. Thus, it's convenient to add an additional task that corresponds to the best tour. It "sends" data to every tree node task, and receives data from some of the leaves. This latter view is convenient for shared-memory, but not so convenient for distributed-memory.

A natural way to agglomerate and map the tasks is to assign a subtree to each thread or process, and have each thread/process carry out all the tasks in its subtree. For example, if we have three threads or processes, as shown earlier in Figure 6.10 , we might map the subtree rooted at $0 \rightarrow 1$ to thread/process 0, the subtree rooted at $0 \rightarrow 2$ to thread/process 1, and the subtree rooted at $0 \rightarrow 3$ to thread/process 2.

Mapping details

There are many possible algorithms for identifying which subtrees we assign to the processes or threads. For example, one thread or process could run the last version of serial depth-first search until the stack stores one partial tour for each thread or process. Then it could assign one tour to each thread or process. The problem with depth-first search is that we expect a subtree whose root is deeper in the tree to require less work than a subtree whose root is higher up in the tree, so we would probably get better load balance if we used something like **breadth-first search** to identify the subtrees.

As the name suggests, breadth-first search searches as widely as possible in the tree before going deeper. So if, for example, we carry out a breadth-first search until we reach a level of the tree that has at least threadcount or commsz nodes, we can then divide the nodes at this level among the threads or processes. See Exercise 6.18 for implementation details.

The best tour data structure

On a shared-memory system, the best tour data structure can be shared. In this setting, the Feasible function can simply examine the data structure. However, updates to the best tour will cause a race condition, and we'll need some sort of locking to prevent errors. We'll discuss this in more detail when we implement the parallel version.

In the case of a distributed-memory system, there are a couple of choices that we need to make about the best tour. The simplest option would be to have the processes operate independently of each other until they have completed searching their subtrees. In this setting, each process would store its own *local* best tour. This local best tour would be used by the process in Feasible and updated by the process each time it calls Updatebesttour. When all the processes have finished searching, they can perform a global reduction to find the tour with the *global* least cost.

This approach has the virtue of simplicity, but it also suffers from the problem that it's entirely possible for a process to spend most or all of its time searching through partial tours that couldn't possibly lead to a global best tour. Thus, we should probably try using an approach that makes the current global best tour available to all the processes. We'll take a look at details when we discuss the MPI implementation.

Dynamic mapping of tasks

A second issue we should consider is the problem of load imbalance. Although the use of breadth-first search ensures that all of our subtrees have approximately the same number of nodes, there is no guarantee that they all have the same amount of work. It's entirely possible that one process or thread will have a subtree consisting of very expensive tours, and, as a consequence, it won't need to search very deeply into its assigned subtree. However, with our current, *static* mapping of tasks to threads/processes, this one thread or process will simply have to wait until the other threads/processes are done.

An alternative is to implement a **dynamic** mapping scheme. In a dynamic scheme, if one thread/process runs out of useful work, it can obtain additional work from another thread/process. In our final implementation of serial depth-first search,

each stack record contains a partial tour. With this data structure a thread or process can give additional work to another thread/process by dividing the contents of its stack. This might at first seem to have the potential for causing problems with the program's correctness, since if we give part of one thread's or one process' stack to another, there's a good chance that the order in which the tree nodes will be visited will be changed.

However, we're already going to do this; when we assign different subtrees to different threads/processes, the order in which the tree nodes are visited is no longer the serial depth-first ordering. In fact, in principle, there's no reason to visit any node before any other node as long as we make sure we visit "ancestors" before "descendants." But this isn't a problem since a partial tour isn't added to the stack until after all its ancestors have been visited. For example, in Figure 6.10 the node consisting of the tour $0 \rightarrow 2 \rightarrow 1$ will be pushed onto the stack when the node consisting of the tour $0 \rightarrow 2$ is the currently active node, and consequently the two nodes won't be on the stack simultaneously. Similarly, the parent of $0 \rightarrow 2$, the root of the tree, 0, is no longer on the stack when $0 \rightarrow 2$ is visited.

A second alternative for dynamic load balancing—at least in the case of shared memory—would be to have a shared stack. However, we couldn't simply dispense with the local stacks. If a thread needed to access the shared stack every time it pushed or popped, there would be a tremendous amount of contention for the shared stack and the performance of the program would probably be worse than a serial program. This is exactly what happened when we parallelized the reduced *n*-body solver with mutexes/locks protecting the calculations of the total forces on the various particles. If every call to `Push` or `Pop` formed a critical section, our program would grind to nearly a complete halt. Thus, we would want to retain local stacks for each thread, with only occasional accesses to the shared stack. We won't pursue this alternative.

See Programming Assignment 6.7 for further details.

A static parallelization of tree search using pthreads

In our static parallelization, a single thread uses breadth-first search to generate enough partial tours so that each thread gets at least one partial tour. Then each thread takes its partial tours and runs iterative tree search on them. We can use the pseudocode shown in Program 6.7 on each thread. Note that most of the function calls—for example, `Besttour`, `Feasible`, `Addcity`—need to access the adjacency matrix representing the digraph, so all the threads will need to access the digraph. However, since these are only *read* accesses, this won't result in a race condition or contention among the threads.

There are only four potential differences between this pseudocode and the pseudocode we used for the second iterative serial implementation:

- The use of mystack instead of stack; since each thread has its own, private
- stack, we use mystack as the identifier for the stack object instead of stack.
- Initialization of the stack.
- Implementation of the Besttour function.

Implementation of the Updatebesttour function.

In the serial implementation, the stack is initialized by pushing the partial tour consisting only of the hometown onto the stack. In the parallel version we need to generate at least threadcount partial tours to distribute among the threads. As we discussed earlier, we can use breadth-first search to generate a list of at least

thread count tours by having a single thread search the tree until it reaches a level with at least thread_count tours. (Note that this implies that the number of threads should be less than $(n-1)!$, which shouldn't be a problem). Then the threads can

```
Partition_tree(my_rank, my_stack);

while (!Empty(my_stack)) {
   curr_tour = Pop(my_stack);
   if (City_count(curr_tour) == n) {
      if (Best_tour(curr_tour)) Update_best_tour(curr_tour);
   } else {
      for (city = n-1; city >= 1; city--)
         if (Feasible(curr_tour, city)) {
            Add_city(curr_tour, city);
            Push_copy(my_stack, curr_tour);
            Remove_last_city(curr_tour)
         }
   }
   Free_tour(curr_tour);
}
```

Program 6.7: Pseudocode for a Pthreads implementation of a statically parallelized solution to TSP

use a block partition to divide these tours among themselves and push them onto their private stacks. Exercise 6.18 looks into the details.

To implement the Besttour function, a thread should compare the cost of its current tour with the cost of the global best tour. Since multiple threads may be simultaneously accessing the global best cost, it might at first seem that there will be a race condition. However, the Besttour function only *reads* the global best cost, so there won't be any conflict with threads that are also checking the best cost. If a thread is updating the global best cost, then a thread that is just checking it will either read the old value or the new, updated value. While we would prefer that it get the new value, we can't insure this without using some very costly locking strategy. For example, threads wanting to execute Besttour or Updatebesttour could wait on a single mutex. This would insure that no thread is updating while another thread is only checking, but would have the unfortunate side effect that only one thread could check the best cost at a time. We could improve on this by using a read-write lock, but this would have the side effect that the readers—the threads calling Besttour—would all block while a thread updated the best tour. In principle, this doesn't sound too bad, but recall that in practice read-write locks can be quite slow. So it seems pretty clear that the "no contention" solution of possibly getting a best tour cost that's out-of-date is probably better, as the next time the thread calls Besttour, it will get the updated value of the best tour cost.

On the other hand, we call Updatebesttour with the intention of *writing* to the best tour structure, and this clearly can cause a race condition if two threads call it simultaneously. To avoid this problem, we can protect the body of the Updatebesttour function with a mutex. This isn't enough, however; between the time a thread completes the test in Besttour and the time it obtains the lock in Updatebesttour, another thread may have obtained the lock and updated the best tour cost, which now may be less than the best tour cost that the first thread found in Besttour. Thus, correct pseudocode for Updatebesttour should look something like this:

```
pthread_mutex_lock(best tour mutex);
/* We've already checked Besttour, but we need to check it again */
if (Besttour(tour))
    Replace old best tour with tour; pthread_mutex_unlock(best tour mutex).
```

This may seem wasteful, but if updates to the best tour are infrequent, then most of the time Besttour will return false and it will only be rarely necessary to make the "double" call.

A dynamic parallelization of tree search using pthreads

If the initial distribution of subtrees doesn't do a good job of distributing the work among the threads, the static parallelization provides no means of redistributing work. The threads with "small" subtrees will finish early, while the threads with large subtrees will continue to work. It's not difficult to imagine that one thread gets the lion's share of the work because the edges in its initial tours are very cheap,

while the edges in the other threads' initial tours are very expensive. To address this issue, we can try to dynamically redistribute the work as the computation proceeds.

To do this, we can replace the test !Empty(mystack) controlling execution of the **while** loop with more complex code. The basic idea is that when a thread runs out of work—that is, !Empty(mystack) becomes false—instead of immediately exiting the **while** loop, the thread waits to see if another thread can provide more work. On the other hand, if a thread that still has work in its stack finds that there is at least one thread without work, and its stack has at least two tours, it can "split" its stack and provide work for one of the threads.

Pthreads condition variables provide a natural way to implement this. When a thread runs out of work it can call pthreadcondwait and go to sleep. When a thread with work finds that there is at least one thread waiting for work, after splitting its stack, it can call pthreadcondsignal. When a thread is awakened it can take one of the halves of the split stack and return to work.

This idea can be extended to handle termination. If we maintain a count of the number of threads that are in pthreadcondwait, then when a thread whose stack is empty finds that threadcount−1 threads are already waiting, it can call pthreadcondbroadcast and as the threads awaken, they'll see that all the threads have run out of work and quit.

Termination

Thus, we can use the pseudocode shown in Program 6.8 for a Terminated function that would be used instead of Empty for the **while** loop implementing tree search.

There are several details that we should look at more closely. Notice that the code executed by a thread before it splits its stack is fairly complicated. In Lines 1–2 the thread

- checks that it has at least two tours in its stack,
- checks that there are threads waiting, and
- checks whether the newstack variable is NULL.

The reason for the check that the thread has enough work should be clear: if there are fewer than two records on the thread's stack, "splitting" the stack will either do nothing or result in the active thread's trading places with one of the waiting threads.

It should also be clear that there's no point in splitting the stack if there aren't any threads waiting for work. Finally, if some thread has already split its stack, but a waiting thread hasn't retrieved the new stack, that is, newstack != NULL, then it would be disastrous to split a stack and overwrite the existing new stack. Note that this makes it essential that after a thread retrieves newstack by, say, copying newstack into its private mystack variable, the thread must set newstack to NULL.

If all three of these conditions hold, then we can try splitting our stack. We can acquire the mutex that protects access to the objects controlling termination (threadsincondwait, newstack, and the condition variable). However, the condition

threads in cond wait > 0 && newstack == NULL

```
1   if (my stack size >= 2 && threads in cond wait > 0 &&
2   newstack == NULL) {
3   lock termmutex;
4   if (threads in cond wait > 0 && newstack == NULL) {
5   Split mystack creating newstack;
6   pthreadcondsignal(&term cond var);
7   }
8   unlock termmutex;
9   return 0; /* Terminated = false; don't quit */ 10 } else if (!Empty(mystack)) /* Keep
    working */

11  return 0; /* Terminated = false; don't quit */
12  } else { /* My stack is empty */
13  lock termmutex;
14  if (threads in cond wait == threadcount−1)

15  /* Last thread running */
16  threads in cond wait++;
17  pthreadcondbroadcast(&term cond var);
18  unlock termmutex;
19  return 1; /* Terminated = true; quit */
20  } else { /* Other threads still working, wait for work */
```

```
21      threads in cond wait++;
22      while (pthreadcondwait(&term cond var, &termmutex)!= 0) ; 23 /* We've been
                                                                  awakened */
24      if (threads in cond wait < threadcount) { /* We got work */
25      mystack = newstack;
26      newstack = NULL;
27      threads in cond wait--;
28      unlock termmutex;
29      return 0; /* Terminated = false */
30      } else { /* All threads done */
31      unlock termmutex;
32      return 1; /* Terminated = true; quit */
33      }
34      } /* else wait for work */
    }   /* else my-stack is empty */
```

35

Program 6.8: Pseudocode for Pthreads Terminated function

can change between the time we start waiting for the mutex and the time we actually acquire it, so as with Updatebesttour, we need to confirm that this condition is still true after acquiring the mutex (Line 4). Once we've verified that these conditions still hold, we can split the stack, awaken one of the waiting threads, unlock the mutex, and return to work.

If the test in Lines 1 and 2 is false, we can check to see if we have any work at all—that is, our stack is nonempty. If it is, we return to work. If it isn't, we'll start the termination sequence by waiting for and acquiring the termination mutex in Line 13.

Once we've acquired the mutex, there are two possibilities:

- We're the last thread to enter the termination sequence, that is,
- threadsincondwait == threadcount-1.

Other threads are still working.

In the first case, we know that since all the other threads have run out of work, and we have also run out of work, the tree search should terminate. We therefore signal all the other threads by calling pthreadcondbroadcast and returning true. Before executing the broadcast, we increment threadsincondwait, even though the broadcast is telling all the threads to return from the condition wait. The reason is that threadsincondwait is serving a dual purpose: When it's less than thread_count, it tells us how many threads are waiting. However, when it's equal to thread_count, it tells us that all the threads are out of work, and it's time to quit.

In the second case—other threads are still working—we call pthreadcondwait (Line 22) and wait to be awakened. Recall that it's possible that a thread could be awakened by some event other than a call to pthreadcondsignal or pthreadcondbroadcast. So, as usual, we put the call to pthreadcondwait in a **while** loop, which will immediately call pthreadcondwait again if some other event (return value not 0) awakens the thread.

Once we've been awakened, there are also two cases to consider:

-
- threadsincondwait < threadcount threads_incond_wait == threadcount

In the first case, we know that some other thread has split its stack and created more work. We therefore copy the newly created stack into our private stack, set the newstack variable to NULL, and decrement threadsincondwait (i.e., Lines 25–27). Recall that when a thread returns from a condition wait, it obtains the mutex associated with the condition variable, so before returning, we also unlock the mutex (i.e., Line 28). In the second case, there's no work left, so we unlock the mutex and return true.

In the actual code, we found it convenient to group the termination variables together into a single struct. Thus, we defined something like

```
typedef struct {
    my_stack_t *newstack; int threads_in_cond_wait; pthread_cond_t term_cond_var;
    pthread_mutex_t termmutex;
} termstruct; typedef termstruct_t *term_t; term_t term; // global variable
```

and we defined a couple of functions, one for initializing the term variable and one for destroying/freeing the variable and its members.

Before discussing the function that splits the stack, note that it's possible that a thread with work can spend a lot of time waiting for termmutex before being able to split its stack. Other threads may be either trying to split their stacks, or preparing for the condition wait. If we suspect that this is a problem, Pthreads provides a nonblocking alternative to pthreadmutexlock called pthreadmutextrylock:

```
int pthreadmutextrylock(pthread_mutex_t *mutex_p /* in/out */);
```

This function attempts to acquire mutexp. However, if it's locked, instead of waiting, it returns immediately. The return value will be zero if the calling thread has successfully acquired the mutex, and nonzero if it hasn't. As an alternative to waiting on the mutex before splitting its stack, a thread can call pthreadmutextrylock. If it acquires termmutex, it can proceed as before. If not, it can just return. Presumably on a subsequent call it can successfully acquire the mutex.

Splitting the stack

Since our goal is to balance the load among the threads, we would like to insure that the amount of work in the new stack is roughly the same as the amount remaining in the original stack. We have no way of knowing in advance of searching the subtree rooted at a partial tour how much work is actually associated with the partial tour, so we'll never be able to guarantee an equal division of work. However, we can use the same strategy that we used in our original assignment of subtrees to threads: that the subtrees rooted at two partial tours with the same number of cities have identical structures. Since on average two partial tours with the same number of cities are equally likely to lead to a "good" tour (and hence

more work), we can try splitting the stack by assigning the tours on the stack on the basis of their numbers of edges. The tour with the least number of edges remains on the original stack, the tour with the next to the least number of edges goes to the new stack, the tour with the next number of edges remains on the original, and so on.

This is fairly simple to implement, since the tours on the stack have an increasing number of edges. That is, as we proceed from the bottom of the stack to the top of the stack, the number of edges in the tours increases. This is because when we push a new partial tour with k edges onto the stack, the tour that's immediately "beneath" it on the stack either has k edges or $k-1$ edges. We can implement the split by starting at the bottom of the stack, and alternately leaving partial tours on the old stack and pushing partial tours onto the new stack, so tour 0 will stay on the old stack, tour 1 will go to the new stack, tour 2 will stay on the old stack, and so on. If the stack is implemented as an array of tours, this scheme will require that the old stack be "compressed" so that the gaps left by removing alternate tours are eliminated. If the stack is implemented as a linked list of tours, compression won't be necessary.

This scheme can be further refined by observing that partial tours with lots of cities won't provide much work, since the subtrees that are rooted at these trees are very small. We could add a "cutoff size" and not reassign a tour unless its number of cities was less than the cutoff. In a shared-memory setting with an array-based stack, reassigning a tour when a stack is split won't increase the cost of the split, since the tour (which is a pointer) will either have to be copied to the new stack or a new location in the old stack. We'll defer exploration of this alternative to Programming Assignment 6.6.

Table 6.8 Run-Times of Pthreads Tree-Search Programs
(times in seconds)

	First Problem			Second Problem		
Threads	*Serial*	*Static*	*Dynamic*	*Serial*	*Static*	*Dynamic*
1	32.9	32.7	34.7	(0)	26.0	25.8
		27.5	(0)			
2	27.9	28.9	(7)	25.8	19.2	(6)
4	25.7	25.9	(47)	25.8	9.3	(49)
8	23.8	22.4	(180)	24.0	5.7	(256)

Evaluating the Pthreads tree-search programs

Table 6.8 shows the performance of the two Pthreads programs on two fifteen-city problems. The "Serial" column gives the run-time of the second iterative solution—the solution that pushes a copy of each new tour onto the stack. For reference, the

first problem in Table 6.8 is the same as the problem the three serial solutions were tested with in Table 6.7, and both the Pthreads and serial implementations were tested on the same system. Run-times are in seconds, and the numbers in parentheses next to the run-times of the program that uses dynamic partitioning give the total number of times the stacks were split.

From these numbers, it's apparent that different problems can result in radically different behaviors. For example, the program that uses static partitioning generally performs better on the first problem than the program that uses dynamic partitioning. However, on the second problem, the performance of the static program is essentially independent of the number of threads, while the dynamic program obtains excellent performance. In general, it appears that the dynamic program is more scalable than the static program.

As we increase the number of threads, we would expect that the size of the local stacks will decrease, and hence threads will run out of work more often. When threads are waiting, other threads will split their stacks, so as the number of threads is increased, the total number of stack splits should increase. Both problems confirm this prediction.

It should be noted that if the input problem has more than one possible solution— that is, different tours with the same minimum cost—then the results of both of the programs are nondeterministic. In the static program, the sequence of best tours depends on the speed of the threads, and this sequence determines which tree nodes are examined. In the dynamic program, we also have nondeterminism because different runs may result in different places where a thread splits its stack and variation in which thread receives the new work. This can also result in run-times, especially dynamic run-times, that are *highly* variable.

6.2.9 Parallelizing the tree-search programs using OpenMP

The issues involved in implementing the static and dynamic parallel tree-search programs using OpenMP are the same as the issues involved in implementing the programs using Pthreads.

There are almost no substantive differences between a static implementation that uses OpenMP and one that uses Pthreads. However, a couple of points should be mentioned:

1. When a single thread executes some code in the Pthreads version, the test

if (my rank == whatever) can be replaced by the OpenMP directive

```
# pragma omp single
```

This will insure that the following structured block of code will be executed by one thread in the team, and the other threads in the team will wait in an implicit barrier at the end of the block until the executing thread is finished.

When whatever is 0 (as it is in each test in the Pthreads program), the test can also be replaced by the OpenMP directive

```
# pragma omp master
```

This will insure that thread 0 executes the following structured block of code. However, the master directive doesn't put an implicit barrier at the end of the block, so it may be necessary to also add a barrier directive after a structured block that has been modified by a master directive.

2. The Pthreads mutex that protects the best tour can be replaced by a single critical directive placed either inside the Updatebesttour function or immediately before the call to Updatebesttour. This is the only potential source of a race condition after the distribution of the initial tours, so the simple critical directive won't cause a thread to block unnecessarily.

The dynamically load-balanced Pthreads implementation depends heavily on Pthreads condition variables, and OpenMP doesn't provide a comparable object. The rest of the Pthreads code can be easily converted to OpenMP. In fact, OpenMP even provides a nonblocking version of ompsetlock. Recall that OpenMP provides a lock object omplockt and the following functions for acquiring and relinquishing the lock, respectively:

void omp set lock(omp lock t*

*   *   *

It also provides the function

int omp test lock(omp lock t* lock p /* *in/out* */) ;

lockp /* *in/out* */) ; **void** omp unset lock(omp lock t lock_p / *in/out* /) ;

which is analogous to pthreadmutextrylock; it attempts to acquire the lock *lockp, and if it succeeds it returns true (or nonzero). If the lock is being used by some other thread, it returns immediately with return value false (or zero).

If we examine the pseudocode for the Pthreads Terminated function in Program 6.8, we see that in order to adapt the Pthreads version to OpenMP, we need to emulate the functionality of the Pthreads function calls

```
pthreadcondsignal(&term cond var); pthreadcondbroadcast(&term cond var);
pthreadcondwait(&term cond var, &termmutex);
```

in Lines 6, 17, and 22, respectively.

Recall that a thread that has entered the condition wait by calling pthreadcondwait(&term cond var, &termmutex); is waiting for either of two events:

- Another thread has split its stack and created work for the waiting thread.
- All of the threads have run out of work.

Perhaps the simplest solution to emulating a condition wait in OpenMP is to use busy-waiting. Since there are two conditions a waiting thread should test for, we can use two different variables in the busy-wait loop:

```
/* Global variables */ int awakenedthread = -1 ;
int workremains = 1; /* true */ . . .
while (awakenedthread != my rank && workremains);
```

Initialization of the two variables is crucial: If awakenedthread has the value of some thread's rank, that thread will exit immediately from the **while**, but there may be no work available. Similarly, if workremains is initialized to 0, all the threads will exit the **while** loop immediately and quit.

Now recall that when a thread enters a Pthreads condition wait, it relinquishes the mutex associated with the condition variable so that another thread can also enter the condition wait or signal the waiting thread. Thus, we should relinquish the lock used in the Terminated function before starting the **while** loop.

Also recall that when a thread returns from a Pthreads condition wait, it reacquires the mutex associated with the condition variable. This is especially important in this setting since if the awakened thread has received work, it will need to access the shared data structures storing the new stack. Thus, our complete emulated condition wait should look something like this:

```
/* Global vars */ int awakenedthread =
-1 ; workremains = 1; /* true */ . . .

omp_unset_lock(&termlock); while (awakenedthread != my_rank &&
workremains); omp_set_lock(&termlock);
```

If you recall the discussion of busy-waiting in Section 4.5 and Exercise 4.3 of Chapter 4, you may be concerned about the possibility that the compiler might reorder the code around the busy-wait loop. The compiler should not reorder across calls to ompsetlock or ompunsetlock. However, the updates to the variables *could* be reordered, so if we're going to be using compiler optimization, we should declare both with the **volatile** keyword.

Emulating the condition broadcast is straightforward: When a thread determines that there's no work left (Line 14 in Program 6.8), then the condition broadcast (Line 17) can be replaced with the assignment workremains = 0; /*

*Assign false to workremains */*

The "awakened" threads can check if they were awakened by some thread's setting workremains to false, and, if they were, return from Terminated with the value true.

Emulating the condition signal requires a little more work. The thread that has split its stack needs to choose one of the sleeping threads and set the variable awakenedthread to the chosen thread's rank. Thus, at a minimum, we need to keep a list of the ranks of the sleeping threads. A simple way to do this is to use a shared queue of thread ranks. When a thread runs out of work, it enqueues its rank before entering the busy-wait loop. When a thread splits its stack, it can choose the thread to awaken by dequeuing the queue of waiting threads:

```
gotlock = omp_test_lock(&termlock); if (gotlock != 0) {
    if (waitingthreads > 0 && newstack == NULL) { Split mystack creating
        newstack;
        awakenedthread = Dequeue(termqueue);
    }
    omp_unset_lock(&termlock);

}
```

The awakened thread needs to reset awakenedthread to −1 before it returns from its call to the Terminated function.

Note that there is no danger that some other thread will be awakened before the awakened thread reacquires the lock. As long as newstack is not NULL, no thread will attempt to split its stack, and hence no thread will try to awaken another thread. So if several threads call Terminated before the awakened thread reacquires the lock, they'll either return if their stacks are nonempty, or they'll enter the wait if their stacks are empty.

Performance of the OpenMP implementations

Table 6.9 shows run-times of the two OpenMP implementations on the same two fifteen-city problems that we used to test the Pthreads implementations. The programs

Table 6.9 Performance of OpenMP and Pthreads Implementations of Tree Search (times in seconds)

	First Problem				**Second Problem**			
	Static		*Dynamic*		*Static*		*Dynamic*	
Th	*OMP*	*Pth*	*OMP*	*Pth*	*OMP*	*Pth*	*OMP*	*Pth*

1	32.5	32.7	33.7	(0)	34.7	(0)	25.6	25.8	
	26.6	(0)	27.5	(0)					
2	27.7	27.9	28.0	(6)	28.9	(7)	25.6	25.8	
	18.8	(9)	19.2	(6)					
4	25.4	25.7	33.1	(75)	25.9	(47)	25.6	25.8	9.8
	(52)	9.3	(49)						
8	28.0	23.8	19.2	(134)	22.4	(180)	23.8	24.0	6.3
	(163)	5.7	(256)						

were also run on the same system we used for the Pthreads and serial tests. For ease of comparison, we also show the Pthreads run-times. Run-times are in seconds and the numbers in parentheses show the total number of times stacks were split in the dynamic implementations.

For the most part, the OpenMP implementations are comparable to the Pthreads implementations. This isn't surprising since the system on which the programs were run has eight cores, and we wouldn't expect busy-waiting to degrade overall performance unless we were using more threads than cores.

There are two notable exceptions for the first problem. The performance of the static OpenMP implementation with eight threads is much worse than the Pthreads implementation, and the dynamic implementation with four threads is much worse than the Pthreads implementation. This could be a result of the nondeterminism of the programs, but more detailed profiling will be necessary to determine the cause with any certainty.

Implementation of tree search using MPI and static partitioning

The vast majority of the code used in the static parallelizations of tree search using Pthreads and OpenMP is taken straight from the second implementation of serial, iterative tree search. In fact, the only differences are in starting the threads, the initial partitioning of the tree, and the `Updatebesttour` function. We might therefore expect that an MPI implementation would also require relatively few changes to the serial code, and this is, in fact, the case.

-

There is the usual problem of distributing the input data and collecting the results. In order to construct a complete tour, a process will need to choose an edge into each vertex and out of each vertex. Thus, each tour will require an entry from each row and each column for each city that's added to the tour, so it would clearly be advantageous for each process to have access to the entire adjacency matrix. Note that the adjacency matrix is going to be relatively small. For example, even if we have 100 cities, it's unlikely that the matrix will require more than 80,000 bytes of storage, so it makes sense to simply read in the matrix on process 0 and broadcast it to all the processes.

Once the processes have copies of the adjacency matrix, the bulk of the tree search can proceed as it did in the Pthreads and OpenMP implementations. The principal differences lie in

- partitioning the tree,
- checking and updating the best tour, and
- after the search has terminated, making sure that process 0 has a copy of the best tour for output.

We'll discuss each of these in turn.

Partitioning the tree

In the Pthreads and OpenMP implementations, thread 0 uses breadth-first search to search the tree until there are at least threadcount partial tours. Each thread then determines which of these initial partial tours it should get and pushes its tours onto its local stack. Certainly MPI process 0 can also generate a list of commsz partial tours. However, since memory isn't shared, it will need to send the initial partial tours to the appropriate process. We could do this using a loop of sends, but distributing the initial partial tours looks an awful lot like a call to MPIScatter. In fact, the only reason we can't use MPIScatter is that the number of initial partial tours may not be evenly divisible by commsz. When this happens, process 0 won't be sending the same number of tours to each process, and MPIScatter requires that the source of the scatter send the same number of objects to each process in the communicator.

Fortunately, there is a variant of MPIScatter, MPIScatterv, which *can* be used to send different numbers of objects to different processes. First recall the syntax of MPIScatter: **int** MPIScatter(

```
                    void int            sendbuf         /* in    */,
                    MPIDatatype         sendcount       /* in    */,
*           *                           sendtype        / in    /,
                    void* recvbuf /* out */, int recvcount /* in */,
                    MPIDatatype recvtype        /* in */, int          root
                        /* in */,
                    MPIComm         comm        /* in */) ;
```

Process root sends sendcount objects of type sendtype from sendbuf to each process in comm. Each process in comm receives recvcount objects of type recvtype into recvbuf. Most of the time, sendtype and recvtype are the same and sendcount and recvcount are also the same. In any case, it's clear that the root process must send the same number of objects to each process. MPIScatterv, on the other hand, has syntax

```
    int             int*        sendcounts          int*    /* in    */,  MPIScatterv(
                    int*        displacements               /* in    */,  void*      sendbuf     /* in */,
*           * MPIDatatype sendtype                          / in    /,
                    void* recvbuf /* out */, int recvcount /* in */,
                    MPIDatatype recvtype        /* in */, int          root
                        /* in */,
                    MPIComm         comm        /* in */) ;
```

The single sendcount argument in a call to MPIScatter is replaced by two array arguments: sendcounts and displacements. Both of these arrays contain commsz elements: sendcounts[q] is the number of objects of type sendtype being sent to process q. Furthermore, displacements[q] specifies the start of the block that is being sent to process q. The displacement is calculated in units of type sendtype. So, for example, if sendtype is MPIINT, and sendbuf has type **int***, then the data that is sent to process q will begin in location sendbuf + displacements[q]

In general, displacements[q] specifies the offset into sendbuf of the data that will go to process q. The "units" are measured in blocks with extent equal to the extent of sendtype.

Similarly, MPIGatherv generalizes MPIGather: **int**

-

void∗ sendbuf **int** sendcount /∗ *in* ∗/,
MPIDatatype sendtype /∗ *in* ∗/,
void∗ recvbuf **int**∗ recvcounts **int**∗ displacements /∗ *in* ∗/, /∗ *out* ∗/, /∗ *in* ∗/,
MPIDatatype recvtype /∗ *in* ∗/,
/ *in* /,

MPIGatherv(

-

* * -
introot /∗ *in* ∗/,
- MPIComm comm /∗ *in* ∗/) ;

Maintaining the best tour

As we observed in our earlier discussion of parallelizing tree search, having each process use its own best tour is likely to result in a lot of wasted computation since the best tour on one process may be much more costly than most of the tours on another process (see Exercise 6.21). Therefore, when a process finds a new best tour, it should send it to the other processes.

First note that when a process finds a new best tour, it really only needs to send its *cost* to the other processes. Each process only makes use of the cost of the current best tour when it calls Besttour. Also, when a process updates the best tour, it doesn't care what the actual cities on the former best tour were; it only cares that the cost of the former best tour is greater than the cost of the new best tour.

-

During the tree search, when one process wants to communicate a new best cost to the other processes, it's important to recognize that we can't use MPIBcast, for recall that MPIBcast is blocking and every process in the communicator must call

MPIBcast. However, in parallel tree search the only process that will know that a broadcast should be executed is the process that has found a new best cost. If it tries to use MPIBcast, it will probably block in the call and never return, since it will be the only process that calls it. We therefore need to arrange that the new tour is sent in such a way that the sending process won't block indefinitely.

MPI provides several options. The simplest is to have the process that finds a new best cost use MPISend to send it to all the other processes:

```
for (dest = 0; dest < comm sz; dest++)
    if (dest != my rank)
        MPISend(&new best cost, 1, MPIINT, dest, NEW COST TAG, comm);
```

Here, we're using a special tag defined in our program, NEWCOSTTAG. This will tell the receiving process that the message is a new cost–as opposed to some other type of message–for example, a tour.

The destination processes can periodically check for the arrival of new best tour costs. We can't use MPIRecv to check for messages since it's blocking; if a process calls

```
MPIRecv(&receivedcost, 1, MPIINT, MPIANY SOURCE, NEW COST TAG, comm, &status);
```

the process will block until a matching message arrives. If no message arrives—for example, if no process finds a new best cost—the process will hang. Fortunately, MPI provides a function that only *checks* to see if a message is available; it doesn't actually try to receive a message. It's called MPIIprobe, and its syntax is

```
int MPIIprobe(
        int int          source          /* in  */,
MPIComm                  tag comm        /* in  */,
                                         /* in  */,  *      *      -
                                         int*     msg    avail  p   -   -
    /* out */,                                                    -   -
    MPIStatus* statusp   /* out */) ;
```

It checks to see if a message from process rank source in communicator comm and with tag tag is available. If such a message is available, *msgavailp will be assigned the value true and the members of *statusp will be assigned the appropriate values. For example, statusp->MPISOURCE will be assigned the rank of the source of the message that's been received. If no message is available, *msgavailp will be assigned the value false. The source and tag arguments can be the wildcards MPIANYSOURCE and MPIANYTAG, respectively. So, to check for a message with a new cost from any process, we can call

```
MPIIprobe(MPI ANY SOURCE, NEW COST TAG, comm, &msgavail, &status);
```

```
MPI_Iprobe(MPI_ANY_SOURCE, NEW_COST_TAG, comm, &msg_avail,
      &status);
while ( msg_avail) {
   MPI_Recv(&received_cost, 1, MPI_INT, status.MPI_SOURCE,
         NEW_COST_TAG, comm, MPI_STATUS_IGNORE);
   if ( received_cost < best_tour_cost)
      best_tour_cost = received_cost;
   MPI_Iprobe(MPI_ANY_SOURCE, NEW_COST_TAG, comm, &msg_avail,
      &status);
} /* while */
```

Program 6.9: MPI code to check for new best tour costs

If msgavail is true, then we can receive the new cost with a call to MPIRecv:

```
MPIRecv(&receivedcost, 1, MPIINT, status.MPISOURCE, NEW COST TAG, comm,
      MPISTATUSIGNORE);
```

A natural place to do this is in the Besttour function. Before checking whether our new tour is the best tour, we can check for new tour costs from other processes with the code in Program 6.9.

This code will continue to receive messages with new costs as long as they're available. Each time a new cost is received that's better than the current best cost, the variable besttourcost will be updated.

Did you spot the potential problem with this scheme? If there is no buffering available for the sender, then the loop of calls to MPISend can cause the sending process to block until a matching receive is posted. If all the other processes have completed their searches, the sending process will hang. The loop of calls to MPISend is therefore unsafe.

There are a couple of alternatives provided by MPI: **buffered sends** and **nonblocking sends**. We'll discuss buffered sends here. See Exercise 6.22 for a discussion of nonblocking operations in MPI.

Modes and Buffered Sends

MPI provides four **modes** for sends: **standard**, **synchronous**, **ready**, and **buffered**. The various modes specify different semantics for the sending functions. The send that we first learned about, MPISend, is the standard mode send. With it, the MPI implementation can decide whether to copy the contents of the message into its own storage or to block until a matching receive is posted. Recall that in synchronous mode, the send will block until a matching receive is posted. In ready mode, the send is erroneous unless a matching receive is posted *before* the send is started. In buffered mode, the MPI implementation must copy the message into local temporary storage if a matching receive hasn't been posted. The local temporary storage must be provided by the user program, not the MPI implementation.

Each mode has a different function: MPISend, MPISsend, MPIRsend, and MPIBsend, respectively, but the argument lists are identical to the argument lists for MPISend:

```
int MPIXsend(
      void*    message   /* in */, int          messagesize   /*
      in */,
```

```
    MPIDatatype message_type /* in */, int dest /* in
    */, int tag  /* in */,
    MPIComm comm      /* in */) ;
```

The buffer that's used by MPIBsend must be turned over to the MPI implementation with a call to MPIBufferattach:

```
int MPIBufferattach(
        void* buffer        /*      in      */,      int
              buffersize /* in */) ;
```

The buffer argument is a pointer to a block of memory allocated by the user program and buffersize is its size in bytes. A previously "attached" buffer can be reclaimed by the program with a call to

```
int MPIBufferdetach(

        void*    buf p      /* out */, int*          buf  size
        p /* out */) ;
```

The *bufp argument returns the address of the block of memory that was previously attached, and *bufsizep gives its size in bytes. A call to MPIBufferdetach will block until all messages that have been stored in the buffer are transmitted. Note that since bufp is an output argument, it should probably be passed in with the ampersand operator. For example:

```
char  buffer[1000];  char*
buf; int bufsize;
. . .
MPIBufferattach(buffer, 1000) ;
. . .
/* Calls to MPIBsend */
. . .
MPIBufferdetach(&buf, &bufsize);
```

At any point in the program only one user-provided buffer can be attached, so if there may be multiple buffered sends that haven't been completed, we need to estimate the amount of data that will be buffered. Of course, we can't know this with any certainty, but we do know that in any "broadcast" of a best tour, the process doing the broadcast will make commsz−1 calls to MPIBsend, and each of these

calls will send a single **int**. We can thus determine the size of the buffer needed for a single broadcast. The amount of storage that's needed for the *data* that's transmitted can be determined with a call to MPIPacksize:

```
int MPI Pack size(
      int        count      /* in */,
                                                  MPIDatatype datatype /* in */,
                                                  MPIComm comm       / in /,
      int*                  size p          /* out */) ;
```

The output argument gives an upper bound on the number of bytes needed to store the data in a message. This won't be enough, however. Recall that in addition to the data, a message stores information such as the destination, the tag, and the communicator, so for each message there is some additional overhead. An upper bound on this additional overhead is given by the MPI constant MPIBSENDOVERHEAD. For a single broadcast, the following code determines the amount of storage needed:

```
int        datasize;        int
messagesize; int bcast buf
size;
```

```
MPI Pack size(1, MPIINT, comm, &datasize); messagesize = datasize
+ MPIBSENDOVERHEAD; bcast buf size = (comm sz -
1)*messagesize;
```

We should guess a generous upper bound on the number of broadcasts and multiply that by bcastbufsize to get the size of the buffer to attach.

Printing the best tour

When the program finishes, we'll want to print out the actual tour as well as its cost, so we do need to get the tour to process 0. It might at first seem that we could arrange this by having each process store its local best tour—the best tour that it finds—and when the tree search has completed, each process can check its local best tour cost and compare it to the global best tour cost. If they're the same, the process could send its local best tour to process 0. There are, however, several problems with this. First, it's entirely possible that there are multiple "best" tours in the TSP digraph, tours that all have the same cost, and different processes may find these different tours. If this happens, multiple processes will try to send their best tours to process 0, and all but one of the threads could hang in a call to MPISend. A

second problem is that it's possible that one or more processes never received the best tour cost, and they may try to send a tour that isn't optimal.

We can avoid these problems by having each process store its local best tour, but after all the processes have completed their searches, they can all call MPIAllreduce and the process with the global best tour can then send it to process 0 for output. The following pseudocode provides some details:

```
struct {
    int cost; int rank; } locdata, globaldata;
locdata.cost = Tourcost(loc best tour); locdata.rank = myrank;
MPIAllreduce(&locdata, &globaldata, 1, MPI2INT, MPIMINLOC, comm);

if (globaldata.rank == 0) return;
    /* 0 already has the best tour */ if (my rank == 0)
    Receive best tour from process globaldata.rank; else if (my rank ==
globaldata.rank) Send best tour to process 0 ;
```

The key here is the operation we use in the call to MPIAllreduce. If we just used MPIMIN, we would know what the cost of the global best tour was, but we wouldn't know who owned it. However, MPI provides a predefined operator, MPIMINLOC, which operates on pairs of values. The first value is the value to be minimized—in our setting, the cost of the tour—and the second value is the *location* of the minimum—in our setting, the rank of the process that actually owns the best tour. If more than one process owns a tour with minimum cost, the location will be the lowest of the ranks of the processes that own a minimum cost tour. The input and the output buffers in the call to MPIAllreduce are two-member structs. Since both the cost and the rank are ints, both members are ints. Note that MPI also provides a predefined type MPI2INT for this type. When the call to MPIAllreduce returns, we have two alternatives:

- If process 0 already has the best tour, we simply return.

- Otherwise, the process owning the best tour sends it to process 0.

Unreceived messages

As we noted in the preceding discussion, it is possible that some messages won't be received during the execution of the parallel tree search. A process may finish searching its subtree before some other process has found a best tour. This won't cause the program to print an incorrect result; the call to MPIAllreduce that finds the process with the best tour won't return until every process has called it, and some process will have the best tour. Thus, it will return with the correct least-cost tour, and process 0 will receive this tour.

However, unreceived messages can cause problems with the call to MPIBufferdetach or the call to MPIFinalize. A process can hang in one of these calls if it is storing buffered messages that were never received, so before we attempt to shut down MPI, we can try to receive any outstanding messages by using MPIIprobe. The code is very similar to the code we used to check for new best tour costs. See Program 6.9. In fact, the only messages that are not sent in collectives are the "best tour" message sent to process 0, and the best tour cost broadcasts. The MPI collectives will hang if some process doesn't participate, so we only need to look for unreceived best tours.

In the dynamically load-balanced code (which we'll discuss shortly) there are other messages, including some that are potentially quite large. To handle this situation, we can use the status argument returned by MPIIprobe to determine the size of the message and allocate additional storage as necessary (see Exercise 6.23).

6.2.12 Implementation of tree search using MPI and dynamic partitioning

In an MPI program that dynamically partitions the search tree, we can try to emulate the dynamic partitioning that we used in the Pthreads and OpenMP programs. Recall that in those programs, before each pass through the main **while** loop in the search function, a thread called a boolean-valued function called Terminated. When a thread ran out of work—that is, its stack was empty—it went into a condition wait (Pthreads) or a busy-wait (OpenMP) until it either received additional work or it was notified that there was no more work. In the first case, it returned to searching for a best tour. In the second case, it quit. A thread that had at least two records on its stack would give half of its stack to one of the waiting threads.

Much of this can be emulated in a distributed-memory setting. When a process runs out of work, there's no condition wait, but it can enter a busy-wait, in which it waits to either receive more work or notification that the program is terminating. Similarly, a process with work can split its stack and send work to an idle process.

The key difference is that there is no central repository of information on which processes are waiting for work, so a process that splits its stack can't just dequeue a queue of waiting processes or call a function such as pthreadcondsignal. It needs to "know" a process that's waiting for work so it can send the waiting process more work. Thus, rather than simply going into a busy-wait for additional work or termination, a process that has run out of work should send a request for work to another process. If it does this, then, when a process enters the Terminated function, it can check to see if there's a request for work from some other process. If there is, and the process that has just entered Terminated has work, it can send part of its stack to the requesting process. If there is a request, and the process has no work available, it can send a rejection. Thus, when we have distributed-memory, pseudocode for our Terminated function can look something like the pseudocode shown in Program 6.10.

Terminated begins by checking on the number of tours that the process has in its stack (Line 1); if it has at least two that are "worth sending," it calls Fulfillrequest (Line 2). Fulfillrequest checks to see if the process has received a request for work. If it has, it splits its stack and sends work to the requesting process. If it hasn't received a request, it just returns. In either case, when it returns from Fulfillrequest it returns from Terminated and continues searching.

If the calling process doesn't have at least two tours worth sending, Terminated calls Sendrejects (Line 5), which checks for any work requests from other processes and sends a "no work" reply to each requesting process. After this, Terminated checks to see if the calling process has any work at all. If it does—that is, if its stack isn't empty—it returns and continues searching.

Things get interesting when the calling process has no work left (Line 9). If there's only one process in the communicator (commsz= 1), then the process returns from Terminated and quits. If there's more than one process, then the process "announces" that it's out of work in Line 11. This is part of the implementation

```
3     return false;  /* Still more work */
4  } else { /* At most 1 available tour */
5     Send rejects();  /* Tell everyone who's requested */
6                      /* work that I have none         */
7     if (!Empty stack(my stack)) {
8        return false;  /* Still more work */
9     } else {  /* Empty stack */
10       if (comm_sz == 1) return true;
11       Out_of_work();
12       work_request_sent = false;
13       while (1) {
14          Clear_msgs();  /* Msgs unrelated to work, termination */
15          if (No_work_left()) {
16             return true;  /* No work left.  Quit */
17          } else if (!work_request sent) {
18             Send_work_request();  /* Request work from someone */
19             work_request_sent = true;
20          } else {
21             Check_for_work(&work_request sent, &work avail);
22             if (work_avail) {
23                Receive_work(my_stack);
24                return false;
25             }
26          }
27       }  /* while */
28    } /* Empty stack */
29 } /* At most 1 available tour */
```

Program 6.10: Terminated function for a dynamically partitioned TSP solver that uses MPI

of a "distributed termination detection algorithm," which we'll discuss shortly. For now, let's just note that the termination detection algorithm that we used with sharedmemory may not work, since it's impossible to guarantee that a variable storing the number of processes that have run out of work is up to date.

Before entering the apparently infinite **while** loop (Line 13), we set the variable workrequestsent to false (Line 12). As its name suggests, this variable tells us whether we've sent a request for work to another process; if we have, we know that we should wait for work or a message saying "no work available" from that process before sending out a request to another process.

-

The **while**(1) loop is the distributed-memory version of the OpenMP busy-wait loop. We are essentially waiting until we either receive work from another process or we receive word that the search has been completed.

When we enter the **while**(1) loop, we deal with any outstanding messages in Line 14. We may have received updates to the best tour cost and we may have received requests for work. It's essential that we tell processes that have requested work that we have none, so that they don't wait forever when there's no work available. It's also a good idea to deal with updates to the best tour cost, since this will free up space in the sending process' message buffer.

After clearing out outstanding messages, we iterate through the possibilities: whether the request has been fulfilled or rejected. If it has been fulfilled, we receive the new work and return to searching. If we received a rejection, we set workrequestsent to false and continue in the loop. If the request was neither fulfilled nor rejected, we also continue in the **while**(1) loop.

- The search has been completed, in which case we quit (Lines 15–16).
- We don't have an outstanding request for work, so we choose a process and send it a request (Lines 17–19). We'll take a closer look at the problem of which process should be sent a request shortly.
- We do have an outstanding request for work (Lines 21–25). So we check

Let's take a closer look at some of these functions.

Myavailtourcount

The function Myavailtourcount can simply return the size of the process' stack. It can also make use of a "cutoff length." When a partial tour has already visited most of the cities, there will be very little work associated with the subtree rooted at the partial tour. Since sending a partial tour is likely to be a relatively expensive operation, it may make sense to only send partial tours with fewer than some cutoff number of edges. In Exercise 6.24 we take a look at how such a cutoff affects the overall run-time of the program.

Fulfillrequest

If a process has enough work so that it can usefully split its stack, it calls Fulfillrequest (Line 2). Fulfillrequest uses MPIIprobe to check for a request for work from another process. If there is a request, it receives it, splits its stack, and sends work to the requesting process. If there isn't a request for work, the process just returns.

-

Splitting the stack

A Splitstack function is called by Fulfillrequest. It uses the same basic algorithm as the Pthreads and OpenMP functions, that is, alternate partial tours with fewer than splitcutoff cities are collected for sending to the process that has requested work. However, in the shared-memory programs, we simply copy the tours (which are pointers) from the original stack to a new stack. Unfortunately, because of the pointers involved in the new stack, such a data structure cannot be simply sent to another process (see Exercise 6.25). Thus, the MPI version of Splitstack *packs* the

contents of the new stack into contiguous memory and sends the block of contiguous memory, which is *unpacked* by the receiver into a new stack.

MPI provides a function, MPIPack, for packing data into a buffer of contiguous memory. It also provides a function, MPIUnpack, for unpacking data from a buffer of contiguous memory. We took a brief look at them in Exercise 6.20 of Chapter 3. Recall that their syntax is

```
int MPIPack(
        void*   data to be packed  /* in */, int   to be packed count
                /* in */, MPIDatatype datatype  /* in */,  void*
                contigbuf  /* out */, int  contig buf size  /*  in
                */, int*  positionp  /* in/out */,
                MPIComm  comm  /* in */) ;

int MPIUnpack(
        void*   contigbuf  /* in */, int  contig buf size  /*  in
                */, int*  positionp  /* in/out */, void*  unpackeddata
                /* out */, int  unpackcount  /* in */,
        MPIDatatype datatype  /* in */,
        MPI_Comm  comm  /* in */) ;
```

MPIPack takes the data in datatobepacked and packs it into contigbuf. The *positionp argument keeps track of where we are in contigbuf. When the function is called, it should refer to the first available location in contigbuf before datatobepacked is added. When the function returns, it should refer to the first available location in contigbuf after datatobepacked has been added.

MPIUnpack reverses the process. It takes the data in contigbuf and unpacks it into unpackeddata. When the function is called, *positionp should refer to the first location in contigbuf that hasn't been unpacked. When it returns, *positionp should refer to the next location in contigbuf after the data that was just unpacked.

As an example, suppose that a program contains the following definitions:

```
typedef struct {
    int* cities; /* Cities in partial tour */ int count; /* Number of cities in
    partial tour */ int cost; /* Cost of partial tour */
} tourstruct; typedef tourstruct* tour_t;
```

Then we can send a variable with type tourt using the following code:

```
void Sendtour(tour_t tour, int dest) {

    int position = 0 ;

        MPIPack(tour->cities, n+1, MPIINT, contigbuf, LARGE,
            &position, comm);
        MPIPack(&tour->count, 1, MPIINT, contigbuf, LARGE,
            &position, comm);
    MPIPack(&tour->cost, 1, MPIINT, contigbuf, LARGE,
            &position, comm);
    MPISend(contigbuf, position, MPIPACKED, dest, 0, comm);
```

```
} /* Sendtour */
```

Similarly, we can receive a variable of type tourt using the following code:

```
void Receivetour(tour t tour, int src) {

    int position = 0 ;

    MPIRecv(contigbuf, LARGE, MPIPACKED, src, 0, comm,
            MPISTATUSIGNORE);
    MPIUnpack(contigbuf, LARGE, &position, tour->cities, n+1,
            MPIINT, comm);
    MPIUnpack(contigbuf, LARGE, &position, &tour->count, 1 ,
            MPIINT, comm);
    MPIUnpack(contigbuf, LARGE, &position, &tour->cost, 1 ,
            MPIINT, comm);
} /* Receivetour */
```

Note that the MPI datatype that we use for sending and receiving packed buffers is MPIPACKED.

Sendrejects

The Sendrejects function (Line 5) is similar to the function that looks for new best tours. It uses MPIIprobe to search for messages that have requested work. Such messages can be identified by a special tag value, for example, WORKREQTAG. When such a message is found, it's received, and a reply is sent indicating that there is no work available. Note that both the request for work and the reply indicating there is no work can be messages with zero elements, since the tag alone informs the receiver of the message's purpose. Even though such messages have no content outside of the envelope, the envelope does take space and they need to be received.

Distributed termination detection

The functions Outōfwōrk and Noworkleft (Lines 11 and 15) implement the termination detection algorithm. As we noted earlier, an algorithm that's modeled on the termination detection algorithm we used in the shared-memory programs will have problems. To see this, suppose each process stores a variable oow, which stores the number of processes that are out of work. The variable is set to 0 when the program starts. Each time a process runs out of work, it sends a message to all the other processes saying it's out of work so that all the processes will increment their copies of oow. Similarly, when a process receives work from another process, it sends a message to every process informing them of this, and each process will decrement its copy of oow. Now suppose we have three process, and process 2 has work but processes 0 and 1 have run out of work. Consider the sequence of events shown in Table 6.10.

The error here is that the work sent from process 1 to process 0 is lost. The reason is that process 0 receives the notification that process 2 is out of work before it receives the notification that process 1 has received work. This may seem improbable,

Table 6.10 Termination Events that Result in an Error

Time	Process 0	Process 1	Process 2

0	Out of Work	Out of Work	Working
	Notify 1, 2	Notify 0, 2	oow = 0 oow = 1 oow = 1
1	Send request to 1	Send Request to 2	Recv notify fr 1 oow = 1 oow = 1 oow = 1
2	oow = 1	Recv notify fr 0	Recv request fr 1
		oow = 2	oow = 1
3	oow = 1	oow = 2	Send work to 1
oow = 0			
4	oow = 1	Recv work fr 2	Recv notify fr 0
		oow = 1	oow = 1
5	oow = 1	Notify 0	Working
		oow = 1	oow = 1
6	oow = 1	Recv request fr 0	Out of work
		oow = 1	Notify 0, 1
oow = 2			
7	Recv notify fr 2	Send work to 0	Send request to 1 oow = 2 oow = 0 oow = 2
8	Recv 1st notify fr 1	Recv notify fr 2	oow = 2 oow = 3 oow = 1
9	Quit	Recv request fr 2	oow = 2 oow = 1

but it's not improbable that process 1 was, for example, interrupted by the operating system and its message wasn't transmitted until after the message from process 2 was transmitted.

Although MPI guarantees that two messages sent from process A to process B will, in general, be received in the order in which they were sent, it makes no guarantee about the order in which messages will be received if they were sent by different processes. This is perfectly reasonable in light of the fact that different processes will, for various reasons, work at different speeds.

Distributed termination detection is a challenging problem, and much work has gone into developing algorithms that are guaranteed to correctly detect it. Conceptually, the simplest of these algorithms relies on keeping track of a quantity that is conserved and can be measured precisely. Let's call it *energy,* since, of course, energy is conserved. At the start of the program, each process has 1 unit of energy. When a process runs out of work, it sends its energy to process 0. When a process fulfills a request for work, it divides its energy in half, keeping half for itself, and sending half to the process that's receiving the work. Since energy is conserved and since the program started with commsz units, the program should terminate when process 0 finds that it has received a total of commsz units.

The Outofwork function when executed by a process other than 0 sends its energy to process 0. Process 0 can just add its energy to a receivedenergy variable. The

Noworkleft function also depends on whether process 0 or some other process is calling. If process 0 is calling, it can receive any outstanding messages sent by Outofwork and add the energy into receivedenergy. If receivedenergy equals commsz, process 0 can send a termination message (with a special tag) to every process. On the other hand, a process other than 0 can just check to see if there's a message with the termination tag.

The tricky part here is making sure that no energy is inadvertently lost; if we try to use **float**s or **double**s, we'll almost certainly run into trouble since at some point dividing by two will result in underflow. Since the amount of energy in exact arithmetic can be represented by a common fraction, we can represent the amount of energy on each process exactly by a pair of fixed-point numbers. The denominator will always be a power of two, so we can represent it by its base-two logarithm. For a large problem it is possible that the numerator could overflow. However, if this becomes a problem, there are libraries that provide arbitrary precision rational numbers (e.g, GMP [21]). An alternate solution is explored in Exercise 6.26.

Sending requests for work

Once we've decided on which process we plan to send a request to, we can just send a zero-length message with a "request for work" tag. However, there are many possibilities for choosing a destination:

1. Loop through the processes in round-robin fashion. Start with (myrank + 1) % commsz and increment this destination (modulo commsz) each time a new request is made. A potential problem here is that two processes can get "in synch" and request work from the same destination repeatedly.

2. Keep a global destination for requests on process 0. When a process runs out of work, it first requests the current value of the global destination from 0. Process 0 can increment this value (modulo commsz) each time there's a request. This avoids the issue of multiple processes requesting work from the same destination, but clearly process 0 can become a bottleneck.

3. Each process uses a random number generator to generate destinations. While it can still happen that several processes may simultaneously request work from the same process, the random choice of successive process ranks should reduce the chance that several processes will make repeated requests to the same process.

These are three possible options. We'll explore these options in Exercise 6.29. Also see [22] for an analysis of the options.

Checking for and receiving work

Once a request is sent for work, it's critical that the sending process repeatedly check for a response from the destination. In fact, a subtle point here is that it's critical that the sending process check for a message from the destination process with a "work available tag" or a "no work available tag." If the sending process simply checks for a message from the destination, it may be "distracted" by other messages from the destination and never receive work that's been sent. For example, there might be a message from the destination requesting work that would mask the presence of a message containing work.

The `Checkforwork` function should therefore first probe for a message from the destination indicating work is available, and, if there isn't such a message, it should probe for a message from the destination saying there's no work available. If there is work available, the `Receivework` function can receive the message with work and unpack the contents of the message buffer into the process' stack. Note also that it needs to unpack the energy sent by the destination process.

Performance of the MPI programs

Table 6.11 shows the performance of the two MPI programs on the same two fifteen-city problems on which we tested the Pthreads and the OpenMP implementations. Run-times are in seconds and the numbers in parentheses show the total number of times stacks were split in the dynamic implementations. These results were obtained on a different system from the system on which we obtained the Pthreads results. We've also included the Pthreads results for this system, so that the two sets of results can be compared. The nodes of this system only have four cores, so the Pthreads results don't include times for 8 or 16 threads. The cutoff number of cities for the MPI runs was 12.

The nodes of this system are small shared-memory systems, so communication through shared variables should be much faster than distributed-memory communication, and it's not surprising that in every instance the Pthreads implementation outperforms the MPI implementation.

The cost of stack splitting in the MPI implementation is quite high; in addition to the cost of the communication, the packing and unpacking is very time-consuming. It's also therefore not surprising that for relatively small problems with few processes, the static MPI parallelization outperforms the dynamic parallelization. However, the

Table 6.11 Performance of MPI and Pthreads Implementations of Tree Search (times in seconds)

	First Problem				Second Problem			
	Static		*Dynamic*		*Static*		*Dynamic*	
Th/Pr	*Pth*	*MPI*	*Pth*	*MPI*	*Pth*	*MPI*	*Pth*	*MPI*
1	35.8	40.9	41.9	(0)	56.5	(0)	27.4	31.5
	32.3	(0)	43.8	(0)				
2	29.9	34.9	34.3	(9)	55.6	(5)	27.4	31.5
	22.0	(8)	37.4	(9)				
4	27.2	31.7	30.2	(55)	52.6	(85)	27.4	31.5
	10.7	(44)	21.8	(76)				
8	35.7	45.5	(165)	35.7	16.5	(161)		
16	20.1	10.5	(441)	17.8	0.1	(173)		

8- and 16-process results suggest that if a problem is large enough to warrant the use of many processes, the dynamic MPI program is much more scalable, and it can provide far superior performance. This is borne out by examination of a 17- city problem run with 16 processes: the dynamic MPI implementation has a run-time of 296 seconds, while the static implementation has a run-time of 601 seconds.

Note that times such as 0.1 second for the second problem running with 16 processes don't really show superlinear speedup. Rather, the initial distribution of work has allowed one of the processes to find the best tour much faster than the initial distributions with fewer processes, and the dynamic partitioning has allowed the processes to do a much better job of load balancing.