An Introduction to Parallel Programming

CHAPTER

Shared-memory programming with Pthreads

4

Recall that from a programmer's point of view, a shared-memory system is one in which all the cores can access all the memory locations (see Fig. 4.1). Thus an obvious approach to the problem of coordinating the work of the cores is to specify that certain memory locations are "shared." This is a very natural approach to parallel programming. Indeed, we might well wonder why all parallel programs don't use this shared-memory approach. However, we'll see in this chapter that there are problems that arise with programming shared-memory systems; problems that are often different from the problems encountered in distributed memory programming.

For example, in Chapter 2 we saw that if different cores attempt to update a single shared-memory location, then the contents of the shared location can be unpredictable. The code that updates the shared location is an example of a *critical section*. We'll see some other examples of critical sections, and we'll learn several methods for controlling access to a critical section.

We'll also learn about other issues and techniques in shared-memory programming. In shared-memory programming, an instance of a program running on a processor is usually called a **thread** (unlike MPI, where it's called a process). We'll learn

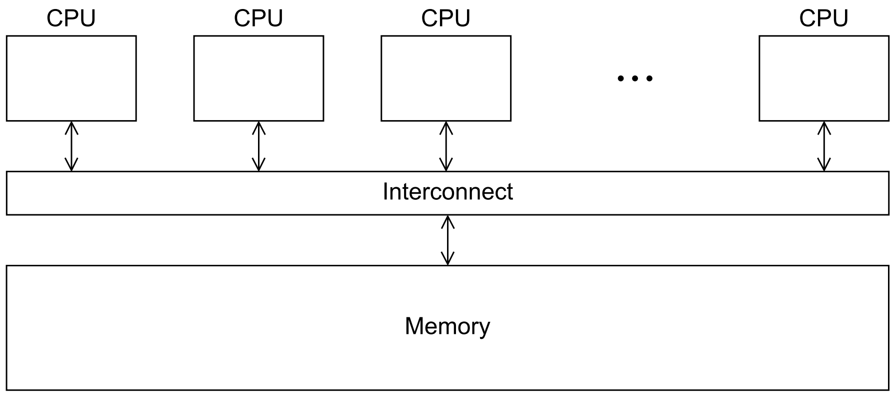

FIGURE 4.1

A shared-memory system.

how to synchronize threads so that each thread will wait to execute a block of statements until another thread has completed some work. We'll learn how to put a thread "to sleep" until a condition has occurred. We'll see that there are some circumstances in which it may at first seem that a critical section must be quite large. However, we'll also see that there are tools that can allow us to "fine-tune" access to these large blocks of code so that more of the program can truly be executed in parallel. We'll see that the use of cache memories can actually cause a shared-memory program to run more slowly. Finally, we'll observe that functions that "maintain state" between successive calls can cause inconsistent or even incorrect results.

In this chapter we'll be using POSIX threads for most of our shared-memory functions. In the next chapter we'll look at an alternative approach to shared-memory programming called OpenMP.

4.1 Processes, threads, and Pthreads

Recall from Chapter 2 that in shared-memory programming, a thread is somewhat analogous to a process in MPI programming. However, it can, in principle, be "lighter-weight." A process is an instance of a running (or suspended) program. In addition to its executable, it consists of the following:

- A block of memory for the stack
- A block of memory for the heap
- Descriptors of resources that the system has allocated for the process—for example, file descriptors (including `stdout`, `stdin`, and `stderr`)
- Security information—for example, information about which hardware and software resources the process can access
- Information about the state of the process, such as whether the process is ready to run or is waiting on a resource, the content of the registers, including the program counter, and so on

In most systems, by default, a process's memory blocks are private: another process can't directly access the memory of a process unless the operating system intervenes. This makes sense. If you're using a text editor to write a program (one process—the running text editor), you don't want your browser (another process) overwriting your text editor's memory. This is even more crucial in a multiuser environment. Ordinarily, one user's processes shouldn't be allowed access to the memory of another user's processes.

However, this isn't desirable when we're running shared-memory programs. At a minimum, we'd like certain variables to be available to multiple processes, allowing much easier memory access. It is also convenient for the processes to share access to things like `stdout` and all other process-specific resources, except for their stacks and program counters. This can be arranged by starting a single process and then having the process start these additional "lighter-weight" processes. For this reason, they're often called **light-weight processes**.

The more commonly used term, **thread**, comes from the concept of "thread of control." A thread of control is just a sequence of statements in a program. The term suggests a stream of control in a single process, and in a shared-memory program a single *process* may have multiple *threads* of control.

As we noted earlier, in this chapter the particular implementation of threads that we'll be using is called POSIX threads or, more often, **Pthreads**. POSIX [46] is a standard for Unix-like operating systems—for example, Linux and macOS. It specifies a variety of facilities that should be available in such systems. In particular, it specifies an application programming interface (API) for *multithreaded* programming.

Pthreads is not a programming language (like C or Java). Rather, like MPI, Pthreads specifies a *library* that can be linked with C programs. Unlike MPI, the Pthreads API is only available on POSIX systems — Linux, macOS, Solaris, HPUX, and so on. Also unlike MPI, there are a number of other widely used specifications for multithreaded programming: Java threads, Windows threads, Solaris threads. However, all of the thread specifications support the same basic ideas, so once you've learned how to program in Pthreads, it won't be difficult to learn how to program with another thread API.

Since Pthreads is a C library, it can, in principle, be used in C++ programs. However, the recent C++11 standard includes its own shared-memory programming model with support for threads (`std::thread`), so it may make sense to use it instead if you're writing C++ programs.

4.2 Hello, world

Let's take a look at a Pthreads program. In Program 4.1, the main function starts up several threads. Each thread prints a message and then quits.

4.2.1 Execution

The program is compiled like an ordinary C program, with the possible exception that we may need to link in the Pthreads library[1]:

```
$ gcc −g −Wall −o pth_hello pth_hello.c −lpthread
```

The `−lpthread` tells the compiler that we want to link in the Pthreads library. Note that it's `−lpthread`, *not* `−lpthreads`. On some systems the compiler will automatically link in the library, and `−lpthread` won't be needed.

1 Recall that the dollar sign (`$`) is the shell prompt, so it shouldn't be typed in. Also recall that for the sake of explicitness, we assume that we're using the Gnu C compiler, `gcc`, and we always use the options `-g`, `-Wall`, and `-o`. See page 77 for further information.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

/* Global variable:  accessible to all threads */
int thread_count;

void *Hello(void* rank);  /* Thread function */

int main(int argc, char* argv[]) {
   long thread;  /* Use long in case of a 64-bit system */
   pthread_t* thread_handles;

   /* Get number of threads from command line */
   thread_count = strtol(argv[1], NULL, 10);

   thread_handles = malloc (thread_count*sizeof(pthread_t));

   for (thread = 0; thread < thread_count; thread++)
      pthread_create(&thread_handles[thread], NULL,
          Hello, (void*) thread);

   printf("Hello from the main thread\n");

   for (thread = 0; thread < thread_count; thread++)
      pthread_join(thread_handles[thread], NULL);

   free(thread_handles);
   return 0;
}  /* main */

void *Hello(void* rank) {
   /* Use long in case of 64-bit system */
   long my_rank = (long) rank;

   printf("Hello from thread %ld of %d\n",
          my_rank, thread_count);

   return NULL;
}  /* Hello */
```

Program 4.1: A Pthreads “hello, world” program.

To run the program, we just type

```
$ ./pth_hello <number of threads>
```

For example, to run the program with 1 thread, we type

```
$ ./pth_hello 1
```

and the output will look something like this:

```
Hello from the main thread
Hello from thread 0 of 1
```

To run the program with four threads, we type

```
$ ./pth_hello 4
```

and the output will look something like this:

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

If your output appears out of order, don't worry. As we will discuss later, we usually do not have direct control of the order in which threads execute.

4.2.2 Preliminaries

Let's take a closer look at the source code in Program 4.1. First notice that this *is* just a C program with a `main` function and one other function. The program includes the familiar `stdio.h` and `stdlib.h` header files. However, there's a lot that's new and different.

In Line 3 we include `pthread.h`, the Pthreads header file, which declares the various Pthreads functions, constants, types, and so on.

In Line 6 we define a *global* variable `thread_count`. In Pthreads programs, global variables are shared by all the threads. Local variables and function arguments—that is, variables declared in functions—are (ordinarily) private to the thread executing the function. If several threads are executing the same function, each thread will have its own private copies of the local variables and function arguments. This makes sense if you recall that each thread has its own stack.

We should keep in mind that global variables can introduce subtle and confusing bugs. For example, suppose we write a program in which we declare a global variable **int** `x`. Then we write a function `f`, in which we intend to use a local variable called `x`, but we forget to declare it. The program will compile with no warnings, since `f` has access to the global `x`. But when we run the program, it produces very strange output, which we eventually determine to have been caused by the fact that the global variable `x` has a strange value. Days later, we finally discover that the strange value came from `f`. As a rule of thumb, we should try to limit our use of global variables to situations in which they're really needed—for example, for a shared variable.

In Line 15 the program gets the number of threads it should start from the command line. Unlike MPI programs, Pthreads programs are typically compiled and run just like serial programs, and one relatively simple way to specify the number of threads that should be started is to use a command-line argument. This isn't a requirement, it's simply a convenient convention we'll be using.

The `strtol` function converts a string into a **long int**. It's declared in `stdlib.h`, and its syntax is

```
long strtol(
          const char*    number_p    /* in    */,
          char**         end_p       /* out   */,
          int            base        /* in    */);
```

It returns a **long int** corresponding to the string referred to by `number_p`. The base of the representation of the number is given by the `base` argument. If `end_p` isn't `NULL`, it will point to the first invalid (that is, nonnumeric) character in `number_p`.

4.2.3 Starting the threads

As we already noted, unlike MPI programs, in which the processes are usually started by a script, in Pthreads the threads are started by the program executable. This introduces a bit of additional complexity, as we need to include code in our program to explicitly start the threads, and we need data structures to store information on the threads.

In Line 17 we allocate storage for one `pthread_t` object for each thread. The `pthread_t` data structure is used for storing thread-specific information. It's declared in `pthread.h`.

The `pthread_t` objects are examples of **opaque** objects. The actual data that they store is system specific, and their data members aren't directly accessible to user code. However, the Pthreads standard guarantees that a `pthread_t` object does store enough information to uniquely identify the thread with which it's associated. So, for example, there is a Pthreads function that a thread can use to retrieve its associated `pthread_t` object, and there is a Pthreads function that can determine whether two threads are in fact the same by examining their associated `pthread_t` objects.

In Lines 19–21, we use the `pthread_create` function to start the threads. Like most Pthreads functions, its name starts with the string `pthread_`. The syntax of `pthread_create` is

```
int pthread_create(
   pthread_t*                thread_p                  /* out */,
   const pthread_attr_t*     attr_p                    /* in  */,
   void*                     (*start_routine)(void*)   /* in  */,
   void*                     args_p                    /* in  */);
```

The first argument is a pointer to the appropriate `pthread_t` object. Note that the object is not allocated by the call to `pthread_create`; it must be allocated *before* the call. We won't be using the second argument, so we just pass `NULL` in our function call.[2] The third argument is the function that the thread is to run, and the last argument is a pointer to the argument that should be passed to the function `start_routine`. The return value for most Pthreads functions indicates if there's been an error in the function call. To reduce the clutter in our examples, in this chapter (as in most of the rest of the book) we'll generally ignore the return values of Pthreads functions.

Let's take a closer look at the last two arguments. The function that's started by `pthread_create` should have a prototype that looks something like this:

```
void* thread_function(void* args_p);
```

Recall that the type **void**∗ can be cast to any pointer type in C, so `args_p` can point to a list containing one or more values needed by `thread_function`. Similarly, the return value of `thread_function` can point to a list of one or more values.

In our call to `pthread_create`, the final argument is a fairly common kluge: we're effectively assigning each thread a unique integer *rank*. Let's first look at why we are doing this; then we'll worry about the details of how to do it.

Consider the following problem: We start a Pthreads program that uses two threads, but one of the threads encounters an error. How do we, the users, know which thread encountered the error? We can't just print out the `pthread_t` object, since it's opaque. However, if when we start the threads, we assign the first thread rank 0, and the second thread rank 1, we can easily determine which thread ran into trouble by just including the thread's rank in the error message.

Since the thread function takes a **void**∗ argument, we could allocate one **int** in `main` for each thread and assign each allocated **int** a unique value. When we start a thread, we could then pass a pointer to the appropriate **int** in the call to `pthread_create`. However, most programmers resort to some trickery with casts. Instead of creating an **int** in `main` for the "rank," we cast the loop variable `thread` to have type **void**∗. Then in the thread function, `hello`, we cast the argument back to a **long** (Line 34).

The result of carrying out these casts is "system-defined," but most C compilers do allow this. However, if the size of pointer types is different from the size of the integer type you use for the rank, you may get a warning. On the machines we used, pointers are 64 bits, and **int**s are only 32 bits, so we use **long** instead of **int**.

Note that our method of assigning thread ranks and, indeed, the thread ranks themselves are just a convenient convention that we'll use. There is no requirement that a thread rank be passed in the call to `pthread_create`, nor a requirement that a thread be assigned a rank. The following thread procedure expects a pointer to a **struct** to be passed in for `args_p`. The **struct** contains both a rank and the name of the task. (Imagine distinguishing between different requests in a web server, for instance.)

[2] Passing `NULL` here uses the default set of Pthread *attributes*—settings that specify a variety of properties, including operating system scheduling parameters and the stack size of the new thread.

```
struct thread_args {
    long my_rank;
    char *task_name;
};

void *Hello(void *args) {
   struct thread_args* t_args
      = (struct thread_args *) args;
   printf("Thread %ld is working on task '%s'\n",
      t_args->my_rank, t_args->task_name);
   return NULL;
}
```

When we create the thread, a pointer to the appropriate **struct** is passed to `pthread_create`. We can add the logic to do this at Line 19 (in this case, each thread has the same "task name"):

```
struct thread_args *t_args
   = malloc(sizeof(struct thread_args));

t_args->my_rank = thread;
t_args->task_name = "Hello task";

pthread_create(&thread_handles[thread],
   NULL,
   Hello,
   (void *) t_args);
```

Also note that there is no technical reason for each thread to run the same function; we could have one thread run `hello`, another run `goodbye`, and so on. However, as with the MPI programs, we'll typically use "single program, multiple data" style parallelism with our Pthreads programs. That is, each thread will run the same thread function, but we'll obtain the effect of different thread functions by branching within a thread.

4.2.4 Running the threads

The thread that's running the `main` function is sometimes called the **main thread**. Hence, after starting the threads, it prints the message

```
Hello from the main thread
```

In the meantime, the threads started by the calls to `pthread_create` are also running. They get their ranks by casting in Line 34, and then print their messages. Note that when a thread is done, since the type of its function has a return value, the thread should return something. In this example, the threads don't actually need to return anything, so they return `NULL`.

As we hinted earlier, in Pthreads the programmer doesn't directly control where the threads are run.[3] There's no argument in `pthread_create` saying which core should run which thread; thread placement is controlled by the operating system. Indeed, on a heavily loaded system, the threads may all be run on the same core. In fact, if a program starts more threads than cores, we should expect multiple threads to be run on a single core. However, if there is a core that isn't being used, operating systems will typically place a new thread on such a core.

4.2.5 Stopping the threads

In Lines 25 and 26, we call the function `pthread_join` once for each thread. A single call to `pthread_join` will wait for the thread associated with the `pthread_t` object to complete. The syntax of `pthread_join` is

```
int pthread_join(
      pthread_t   thread      /* in   */,
      void**      ret_val_p   /* out  */);
```

The second argument can be used to receive any return value computed by the thread. In the example, each thread returns `NULL`, and eventually the main thread will call `pthread_join` on that thread to complete its termination.

This function is called `pthread_join` because of a diagramming style that is often used to describe the threads in a multithreaded process. If we think of the main thread as a single line in our diagram, then, when we call `pthread_create`, we can create a *branch* or *fork* off the main thread. Multiple calls to `pthread_create` will result in multiple branches or forks. Then, when the threads started by `pthread_create` terminate, the diagram shows the branches *joining* the main thread. See Fig. 4.2.

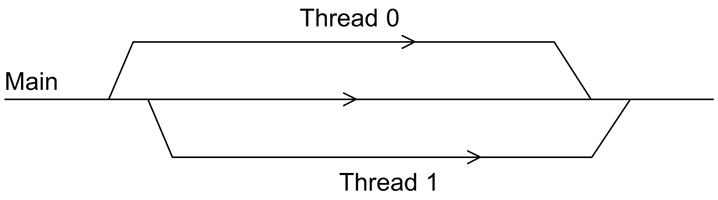

FIGURE 4.2

Main thread forks and joins two threads.

As noted previously, every thread requires a variety of resources to be allocated, including stacks and local variables. The `pthread_join` function not only allows us to wait for a particular thread to finish its execution but also frees the resources associated with the thread. In fact, not joining threads that have finished execution produces

[3] Some systems (for example, some implementations of Linux) do allow the programmer to specify where a thread is run. However, these constructions will not be portable.

zombie threads that waste resources and may even prevent the creation of new threads if left unchecked. If your program does not need to wait for a particular thread to finish, it can be *detached* with the `pthread_detach` function to indicate that its resources should be freed automatically upon termination. See Exercise 4.7 for an example of using `pthread_detach`.

4.2.6 Error checking

In the interest of keeping the program compact and easy to read, we have resisted the temptation to include many details that would be important in a "real" program. The most likely source of problems in this example (and in many programs) is the user input (or lack thereof). Therefore it would be a very good idea to check that the program was started with command line arguments, and, if it was, to check the actual value of the number of threads to see if it's reasonable. If you visit the book's website, you can download a version of the program that includes this basic error checking.

In general, it is good practice to always check the error codes returned by the Pthreads functions. This can be especially useful when you're just starting to use Pthreads and some of the details of function use aren't completely clear. We'd suggest getting in the habit of consulting the "RETURN VALUE" sections of the man pages for Pthreads functions (for instance, see `man pthread_create`; you will note several return values that indicate a variety of errors).

4.2.7 Other approaches to thread startup

In our example, the user specifies the number of threads to start by typing in a command-line argument. The main thread then creates all of the "subsidiary" threads. While the threads are running, the main thread prints a message, and then waits for the other threads to terminate. This approach to threaded programming is very similar to our approach to MPI programming, in which the MPI system starts a collection of processes and waits for them to complete.

There is, however, a very different approach to the design of multithreaded programs. In this approach, subsidiary threads are only started as the need arises. As an example, imagine a Web server that handles requests for information about highway traffic in the San Francisco Bay Area. Suppose that the main thread receives the requests and subsidiary threads fulfill the requests. At 1 o'clock on a typical Tuesday morning, there will probably be very few requests, while at 5 o'clock on a typical Tuesday evening, there will probably be thousands. Thus a natural approach to the design of this Web server is to have the main thread start subsidiary threads when it receives requests.

Intuitively, thread startup involves some overhead. The time required to start a thread will be much greater than, for instance, a floating point arithmetic operation, so in applications that need maximum performance the "start threads as needed" approach may not be ideal. In such a case, it is usually more performant to employ a scheme that leverages the strengths of both approaches: our main thread will start all the threads it anticipates needing at the beginning of the program, but the threads will sit idle instead of terminating when they finish their work. Once another request arrives, an idle thread can fulfill it without incurring thread creation overhead. This approach is called a *thread pool*, which we'll cover in Programming Assignment 4.5.

4.3 Matrix-vector multiplication

Let's take a look at writing a Pthreads matrix-vector multiplication program. Recall that if $A = (a_{ij})$ is an $m \times n$ matrix and $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})^T$ is an n-dimensional column vector,[4] then the matrix-vector product $A\mathbf{x} = \mathbf{y}$ is an m-dimensional column vector, $\mathbf{y} = (y_0, y_1, \ldots, y_{m-1})^T$, in which the ith component y_i is obtained by finding the dot product of the ith row of A with $\mathbf{x}$:

$$y_i = \sum_{j=0}^{n-1} a_{ij} x_j .$$

(See Fig. 4.3.)

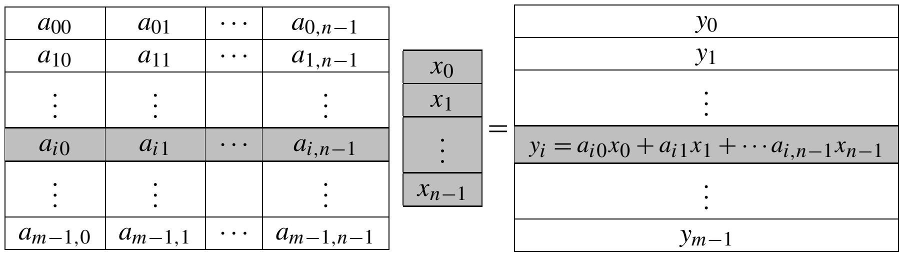
a_{00} a_{01} $\cdots$ $a_{0,n-1}$
a_{10} a_{11} $\cdots$ $a_{1,n-1}$
a_{i0} a_{i1} $\cdots$ $a_{i,n-1}$
$a_{m-1,0}$ $a_{m-1,1}$ $\cdots$ $a_{m-1,n-1}$
x_0 x_1 x_{n-1}
=
y_0 y_1
$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}$
y_{m-1}

FIGURE 4.3

Matrix-vector multiplication.

Thus pseudocode for a *serial* program for matrix-vector multiplication might look like this:

```
/* For each row of A */
for (i = 0; i < m; i++) {
   y[i] = 0.0;
   /* For each element of the row and each element of x */
   for (j = 0; j < n; j++)
      y[i] += A[i][j]* x[j];
}
```

[4] Recall that we use the convention that matrix and vector subscripts start with 0. Also recall that if **b** is a matrix or a vector, then $\mathbf{b}^T$ denotes its transpose.

We want to parallelize this by dividing the work among the threads. One possibility is to divide the iterations of the outer loop among the threads. If we do this, each thread will compute some of the components of y. For example, suppose that $m = n = 6$ and the number of threads, thread_count or t, is three. Then the computation could be divided among the threads as follows:

Thread	**Components of y**
0	y[0], y[1]
1	y[2], y[3]
2	y[4], y[5]

To compute y[0], thread 0 will need to execute the code

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

Therefore thread 0 will need to access every element of row 0 of A and every element of x. More generally, the thread that has been assigned y[i] will need to execute the code

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

Thus this thread will need to access every element of row i of A and every element of x. We see that each thread needs to access every component of x, while each thread only needs to access its assigned rows of A and assigned components of y. This suggests that, at a minimum, x should be shared. Let's also make A and y shared. This might seem to violate our principle that we should only make variables global that need to be global. However, in the exercises, we'll take a closer look at some of the issues involved in making the A and y variables local to the thread function, and we'll see that making them global can make good sense. At this point, we'll just observe that if they are global, the main thread can easily initialize all of A by just reading its entries from stdin, and the product vector y can be easily printed by the main thread.

Having made these decisions, we only need to write the code that each thread will use for deciding which components of y it will compute. To simplify the code, let's assume that both m and n are evenly divisible by t. Our example with $m = 6$ and $t = 3$ suggests that each thread gets m/t components. Furthermore, thread 0 gets the first m/t, thread 1 gets the next m/t, and so on. Thus the formulas for the components assigned to thread q might be

$$\text{first component: } q \times \frac{m}{t}$$

and

$$\text{last component: } (q+1) \times \frac{m}{t} - 1.$$

With these formulas, we can write the thread function that carries out matrix-vector multiplication. (See Program 4.2.) Note that in this code, we're assuming that `A`, `x`, `y`, `m`, and `n` are all global and shared.

```
void *Pth_mat_vect(void* rank) {
   long my_rank = (long) rank;
   int i, j;
   int local_m = m/thread_count;
   int my_first_row = my_rank*local_m;
   int my_last_row = (my_rank+1)*local_m - 1;

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

   return NULL;
}  /* Pth_mat_vect */
```

Program 4.2: Pthreads matrix-vector multiplication.

If you have already read the MPI chapter, you may recall that it took more work to write a matrix-vector multiplication program using MPI. This was because of the fact that the data structures were necessarily distributed, that is, each MPI process only has direct access to its own local memory. Thus for the MPI code, we need to explicitly *gather* all of `x` into each process's memory. We see from this example that there are instances in which writing shared-memory programs is easier than writing distributed-memory programs. However, we'll shortly see that there are situations in which shared-memory programs can be more complex.

4.4 Critical sections

Matrix-vector multiplication was very easy to code, because the shared-memory locations were accessed in a highly desirable way. After initialization, all of the variables—except `y`—are only *read* by the threads. That is, except for `y`, none of the shared variables are changed after they've been initialized by the main thread. Furthermore, although the threads do make changes to `y`, only one thread makes changes to any individual component, so there are no attempts by two (or more) threads to modify any single component. What happens if this isn't the case? That is, what happens when multiple threads update a single memory location? We also discuss this in Chapters 2 and 5, so if you've read one of these chapters, you already know the answer. But let's look at an example.

Let's try to estimate the value of π. There are lots of different formulas we could use. One of the simplest is

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots\right).$$

This isn't the best formula for computing π, because it takes *a lot* of terms on the right-hand side before it is very accurate. However, for our purposes, lots of terms will be better to demonstrate the effects of parallelism.

The following *serial* code uses this formula:

```
double factor = 1.0;
double sum = 0.0;
for (i = 0; i < n; i++, factor = −factor) {
   sum += factor/(2*i+1);
}
pi = 4.0*sum;
```

We can try to parallelize this in the same way we parallelized the matrix-vector multiplication program: divide up the iterations in the **for** loop among the threads and make `sum` a shared variable. To simplify the computations, let's assume that the number of threads, `thread_count` or t, evenly divides the number of terms in the sum, n. Then, if $\bar{n} = n/t$, thread 0 can add the first $\bar{n}$ terms. Therefore for thread 0, the loop variable `i` will range from 0 to $\bar{n} - 1$. Thread 1 will add the next $\bar{n}$ terms, so for thread 1, the loop variable will range from $\bar{n}$ to $2\bar{n} - 1$. More generally, for thread q the loop variable will range over

$$q\bar{n}, q\bar{n}+1, q\bar{n}+2, \ldots, (q+1)\bar{n}-1.$$

Furthermore, the sign of the first term, term $q\bar{n}$, will be positive if $q\bar{n}$ is even and negative if $q\bar{n}$ is odd. The thread function might use the code shown in Program 4.3.

If we run the Pthreads program with two threads and n is relatively small, we find that the results of the Pthreads program are in agreement with the serial sum program. However, as n gets larger, we start getting some peculiar results. For example, with a dual-core processor we get the following results:

	n			
	10^5	**10^6**	**10^7**	**10^8**
π	3.14159	3.141593	3.1415927	3.14159265
1 Thread	3.14158	3.141592	3.1415926	3.14159264
2 Threads	3.14158	3.141480	3.1413692	3.14164686

Notice that as we increase n, the estimate with one thread gets better and better. In fact, with each factor of 10 increase in n, we get another correct digit. With $n = 10^5$, the result as computed by a single thread has five correct digits. With $n = 10^6$, it has six correct digits, and so on. The result computed by two threads agrees with the

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)  /* my_first_i is even */
      factor = 1.0;
   else  /* my_first_i is odd */
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      sum += factor/(2*i+1);
   }

   return NULL;
}  /* Thread_sum */
```

Program 4.3: An attempt at a thread function for computing π.

result computed by one thread when $n = 10^5$. However, for larger values of n, the result computed by two threads actually gets worse. In fact, if we ran the program several times with two threads and the same value of n, we would see that the result computed by two threads *changes* from run to run. The answer to our original question must clearly be, "Yes, it matters if multiple threads try to update a single shared variable."

Let's recall why this is the case. Remember that the addition of two values is typically *not* a single machine instruction. For example, although we can add the contents of a memory location `y` to a memory location `x` with a single C statement,

```
x = x + y;
```

what the machine does is typically more complicated. The current values stored in `x` and `y` will, in general, be stored in the computer's main memory, which has no circuitry for carrying out arithmetic operations. Before the addition can be carried out, the values stored in `x` and `y` may therefore have to be transferred from main memory to registers in the CPU. Once the values are in registers, the addition can be carried out. After the addition is completed, the result may have to be transferred from a register back to memory.

Suppose that we have two threads, and each computes a value that is stored in its private variable `y`. Also suppose that we want to add these private values together into a shared variable `x` that has been initialized to 0 by the main thread. Each thread will execute the following code:

```
y = Compute(my_rank);
x = x + y;
```

Let's also suppose that thread 0 computes `y` = 1 and thread 1 computes `y` = 2. The "correct" result should then be `x` = 3. Here's one possible scenario:

Time	Thread 0	Thread 1
1	Started by main thread	
2	Call `Compute()`	Started by main thread
3	Assign `y = 1`	Call `Compute()`
4	Put `x=0` and `y=1` into registers	Assign `y = 2`
5	Add 0 and 1	Put `x=0` and `y=2` into registers
6	Store 1 in memory location `x`	Add 0 and 2
7		Store 2 in memory location `x`

So we see that if thread 1 copies `x` from memory to a register *before* thread 0 stores its result, the computation carried out by thread 0 will be *overwritten* by thread 1. The problem could be reversed: if thread 1 *races* ahead of thread 0, then its result may be overwritten by thread 0. In fact, unless one of the threads stores its result *before* the other thread starts reading `x` from memory, the "winner's" result will be overwritten by the "loser."

This example illustrates a fundamental problem in shared-memory programming: when multiple threads attempt to update a shared resource—in our case a shared variable—the result may be unpredictable. Recall that more generally, when multiple threads attempt to access a shared resource, such as a shared variable or a shared file, at least one of the accesses is an update, and the accesses can result in an error, we have a **race condition**. In our example, in order for our code to produce the correct result, we need to make sure that once one of the threads starts executing the statement `x = x + y`, it finishes executing the statement *before* the other thread starts executing the statement. Therefore the code `x = x + y` is a **critical section**. That is, it's a block of code that updates a shared resource that can only be updated by one thread at a time.

To further illustrate the concept of a race condition, imagine a bank wants to improve the performance of its checking account system. An obvious first step would be to make the system multithreaded; rather than processing a single transaction at a time, banking operations should be spread across multiple threads to take advantage of parallelism. This works well—until multiple transactions modify an account at the same time. Consider two pending transactions on a checking account with an initial balance of $1000:

- A $100 utility bill payment
- A $500 salary deposit

After the transactions complete, the new account balance should be $1400. The salary deposit will require an addition operation and the utility payment will require a subtraction. However, as mentioned previously, these simple math operations will be broken into more than one machine instruction. One possible outcome is:

Time	Thread 0 (Bill Payment)	Thread 1 (Salary Deposit)
1		Read Balance ($1000)
2	Read Balance ($1000)	Calculate Balance + $500
3	Calculate Balance - $100	Write Balance ($1500)
4	Write Balance ($900)	

Rather than the expected ending balance of $1400, we get $900 instead, because the transaction processed by thread 1 was overwritten by thread 0.

These types of issues are particularly difficult to debug, because the outcome is non-deterministic. It is entirely possible that the error shown above occurs less than 1% of the time and could be influenced by external factors, including the hardware, operating system, or process scheduling algorithm. Even worse, attaching a debugger or adding `printf` statements to the code may change the relative timing of the threads and seemingly "correct" the issue temporarily. Such bugs that disappear when inspected are known as *Heisenbugs* (the act of observing the system alters its state).

4.5 Busy-waiting

To avoid race conditions, threads need exclusive access to shared memory regions. When, say, thread 0 wants to execute the statement `x = x + y`, it needs to first make sure that thread 1 is not already executing the statement. Once thread 0 makes sure of this, it needs to provide some way for thread 1 to determine that it, thread 0, is executing the statement, so that thread 1 won't attempt to start executing the statement until thread 0 is done. Finally, after thread 0 has completed execution of the statement, it needs to provide some way for thread 1 to determine that it is done, so that thread 1 can safely start executing the statement.

A simple approach that doesn't involve any new concepts is the use of a flag variable. Suppose `flag` is a shared **int** that is set to 0 by the main thread. Further, suppose we add the following code to our example:

```
1   y = Compute(my_rank);
2   while (flag != my_rank);
3   x = x + y;
4   flag++;
```

Let's suppose that thread 1 finishes the assignment in Line 1 before thread 0. What happens when it reaches the **while** statement in Line 2? If you look at the **while** statement for a minute, you'll see that it has the somewhat peculiar property that its body is empty. So if the test `flag != my_rank` is true, then thread 1 will just execute the test a second time. In fact, it will keep re-executing the test until the test is false. When the test is false, thread 1 will go on to execute the code in the critical section `x = x + y`.

Since we're assuming that the main thread has initialized `flag` to 0, thread 1 won't proceed to the critical section in Line 3 until thread 0 executes the statement `flag++`. In fact, we see that unless some catastrophe befalls thread 0, it will eventually catch up to thread 1. However, when thread 0 executes its first test of `flag != my_rank`, the condition is false, and it will go on to execute the code in the critical section `x = x + y`. When it's done with this, we see that it will execute `flag++`, and thread 1 can finally enter the critical section.

The key here is that thread 1 *cannot enter the critical section until thread 0 has completed the execution of* `flag++`. And, provided the statements are executed exactly as they're written, this means that thread 1 cannot enter the critical section until thread 0 has completed it.

The **while** loop is an example of **busy-waiting**. In busy-waiting, a thread repeatedly tests a condition, but, effectively, does no useful work until the condition has the appropriate value (false in our example).

Note that we said that the busy-wait solution would work "provided the statements are executed exactly as they're written." If compiler optimization is turned on, it *is* possible that the compiler will make changes that will affect the correctness of busy-waiting. The reason for this is that the compiler is unaware that the program is multithreaded, so it doesn't "know" that the variables `x` and `flag` can be modified by another thread. For example, if our code

```
y = Compute(my_rank);
while (flag != my_rank);
x = x + y;
flag++;
```

is run by just one thread, the order of the statements **while** `(flag != my_rank)` and `x = x + y` is unimportant. An optimizing compiler might therefore determine that the program would make better use of registers if the order of the statements were switched. Of course, this will result in the code

```
y = Compute(my_rank);
x = x + y;
while (flag != my_rank);
flag++;
```

which defeats the purpose of the busy-wait loop. The simplest solution to this problem is to turn compiler optimizations off when we use busy-waiting. For an alternative to completely turning off optimizations, see Exercise 4.3.

We can immediately see that busy-waiting is not an ideal solution to the problem of controlling access to a critical section. Since thread 1 will execute the test over and over until thread 0 executes `flag++`, if thread 0 is delayed (for example, if the operating system preempts it to run something else), thread 1 will simply "spin" on the test, eating up CPU cycles. This approach—often called a *spinlock*—can be positively disastrous for performance. Turning off compiler optimizations can also seriously degrade performance.

Before going on, though, let's return to our π calculation program in Program 4.3 and correct it by using busy-waiting. The critical section in this function is Line 15. We can therefore precede this with a busy-wait loop. However, when a thread is done with the critical section, if it simply increments `flag`, eventually `flag` will be greater than t, the number of threads, and none of the threads will be able to return to the critical section. That is, after executing the critical section once, all the threads will be stuck forever in the busy-wait loop. Thus, in this instance, we don't want to simply increment `flag`. Rather, the last thread, thread $t-1$, should reset `flag` to zero. This can be accomplished by replacing `flag++` with

```
flag = (flag + 1) % thread_count;
```

With this change, we get the thread function shown in Program 4.4. If we compile the program and run it with two threads, we see that it is computing the correct results.

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      while (flag != my_rank);
      sum += factor/(2*i+1);
      flag = (flag+1) % thread_count;
   }

   return NULL;
}  /* Thread_sum */
```

Program 4.4: Pthreads global sum with busy-waiting.

However, if we add in code for computing elapsed time, we see that when $n = 10^8$, the serial sum is consistently faster than the parallel sum. For example, on the dual-core system, the elapsed time for the sum as computed by two threads is about 19.5 seconds, while the elapsed time for the serial sum is about 2.8 seconds!

Why is this? Of course, there's overhead associated with starting up and joining the threads. However, we can estimate this overhead by writing a Pthreads program in which the thread function simply returns:

```
void* Thread_function(void* ignore) {
   return NULL;
}  /* Thread_function */
```

When we find the time that's elapsed between starting the first thread and joining the second thread, we see that on this particular system, the overhead is less than 0.3 milliseconds, so the slowdown isn't due to thread overhead. If we look closely at the thread function that uses busy-waiting, we see that the threads alternate between executing the critical section code in Line 16. Initially `flag` is 0, so thread 1 must wait until thread 0 executes the critical section and increments `flag`. Then, thread 0 must wait until thread 1 executes and increments. The threads will alternate between waiting and executing, and evidently the waiting and the incrementing increase the overall run-time by a factor of seven.

As we'll see, busy-waiting isn't the only solution to protecting a critical section. In fact, there are much better solutions. However, since the code in a critical section can only be executed by one thread at a time, no matter how we limit access to the critical section, we'll effectively serialize the code in the critical section. Therefore, if it's at all possible, we should minimize the number of times we execute critical section code. One way to greatly improve the performance of the sum function is to have each thread use a *private* variable to store its total contribution to the sum. Then, each thread can add in its contribution to the global sum once, *after* the **for** loop. See Program 4.5. When we run this on the dual core system with $n = 10^8$, the elapsed time is reduced to 1.5 seconds for two threads, a *substantial* improvement.

4.6 Mutexes

Since a thread that is busy-waiting may continually use the CPU, busy-waiting is generally not an ideal solution to the problem of limiting access to a critical section. Two better solutions are mutexes and semaphores. **Mutex** is an abbreviation of *mutual exclusion*, and a mutex is a special type of variable that, together with a couple of special functions, can be used to restrict access to a critical section to a single thread at a time. Thus a mutex can be used to guarantee that one thread "excludes" all other threads while it executes the critical section. Hence, the mutex guarantees mutually exclusive access to the critical section.

The Pthreads standard includes a special type for mutexes: `pthread_mutex_t`. A variable of type `pthread_mutex_t` needs to be initialized by the system before it's used. This can be done with a call to

```
int pthread_mutex_init(
      pthread_mutex_t*            mutex_p   /* out */,
      const pthread_mutexattr_t*  attr_p    /* in  */);
```

We won't make use of the second argument, so we'll just pass in `NULL` to use the default attributes. You may also occasionally encounter the following *static* mutex initialization that declares a mutex and initializes it in a single line of code:

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor, my_sum = 0.0;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor)
      my_sum += factor/(2*i+1);

   while (flag != my_rank);
   sum += my_sum;
   flag = (flag+1) % thread_count;

   return NULL;
}  /* Thread_sum */
```

Program 4.5: Global sum function with critical section after loop.

```
   pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Although in general `pthread_mutex_init` is more flexible, this initialization is fine in many, if not most, cases.

When a Pthreads program finishes using a mutex (regardless of how they are initialized), it should call

```
   int pthread_mutex_destroy(
      pthread_mutex_t*  mutex_p  /* in/out */);
```

The point of a mutex is to protect a critical section from being entered by more than one thread at a time. To gain access to a critical section, a thread will lock the mutex, do its work, and then unlock the mutex to let other threads execute the critical section. To lock the mutex and gain exclusive access to the critical section, a thread calls

```
   int pthread_mutex_lock(
      pthread_mutex_t*  mutex_p  /* in/out */);
```

When a thread is finished executing the code in a critical section, it should call

```
   int pthread_mutex_unlock(
      pthread_mutex_t*  mutex_p  /* in/out */);
```

The call to pthread_mutex_lock will cause the thread to wait until no other thread is in the critical section, and the call to pthread_mutex_unlock notifies the system that the calling thread has completed execution of the code in the critical section.

We can use mutexes instead of busy-waiting in our global sum program by declaring a global mutex variable, having the main thread initialize it, and then, instead of busy-waiting and incrementing a flag, the threads call pthread_mutex_lock before entering the critical section, and they call pthread_mutex_unlock when they're done with the critical section. (See Program 4.6.) The first thread to call pthread_mutex_lock

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;
   double my_sum = 0.0;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      my_sum += factor/(2*i+1);
   }
   pthread_mutex_lock(&mutex);
   sum += my_sum;
   pthread_mutex_unlock(&mutex);

   return NULL;
}  /* Thread_sum */
```

Program 4.6: Global sum function that uses a mutex.

will, effectively, "lock the door" to the critical section: any other thread that attempts to execute the critical section code must first also call pthread_mutex_lock, and until the first thread calls pthread_mutex_unlock, all the threads that have called pthread_mutex_lock will **block** in their calls—they'll just wait until the first thread is done. After the first thread calls pthread_mutex_unlock, the system will choose one of the blocked threads and allow it to execute the code in the critical section. This process will be repeated until all the threads have completed executing the critical section.

"Locking" and "unlocking" the door to the critical section isn't the only metaphor that's used in connection with mutexes. Programmers often say that the thread that has returned from a call to `pthread_mutex_lock` has "obtained the mutex" or "obtained the lock." When this terminology is used, a thread that calls `pthread_mutex_unlock` relinquishes the mutex or lock. (You may also encounter terminology referring to this as "acquiring" and "releasing" the lock.)

Notice that with mutexes (unlike our busy-waiting solution), the order in which the threads execute the code in the critical section is more or less random: the first thread to call `pthread_mutex_lock` will be the first to execute the code in the critical section. Subsequent accesses will be scheduled by the system. Pthreads doesn't guarantee that the threads will obtain the lock in the order in which they called `Pthread_mutex_lock`. However, in our setting, a finite number of threads will try to acquire the lock and they are guaranteed to eventually obtain it.

If we look at the (unoptimized) performance of the busy-wait π program (with the critical section after the loop) and the mutex program, we see that for both versions the ratio of the run-time of the single-threaded program with the multithreaded program is equal to the number of threads, as long as the number of threads is no greater than the number of cores. That is,

$$\frac{T_{\text{serial}}}{T_{\text{parallel}}} \approx \texttt{thread\_count},$$

provided `thread_count` is less than or equal to the number of cores. Recall that $T_{\text{serial}}/T_{\text{parallel}}$ is called the *speedup*, and when the speedup is equal to the number of threads, we have achieved more or less "ideal" performance or *linear speedup*.

If we compare the performance of the version that uses busy-waiting with the version that uses mutexes, we don't see much difference in the overall run-time when the programs are run with fewer threads than cores. This shouldn't be surprising, as each thread only enters the critical section once; unless the critical section is very long, or the Pthreads functions are very slow, we wouldn't expect the threads to be delayed very much by waiting to enter the critical section. However, if we start increasing the number of threads beyond the number of cores, the performance of the version that uses mutexes remains largely unchanged, while the performance of the busy-wait version degrades. (See Table 4.1.)

We see that when we use busy-waiting, performance can degrade if there are more threads than cores.[5] This should make sense. For example, suppose we have two cores and five threads. Also suppose that thread 0 is in the critical section, thread 1 is in the busy-wait loop, and threads 2, 3, and 4 have been descheduled by the operating system. After thread 0 completes the critical section and sets `flag` = 1, it will be terminated, and thread 1 can enter the critical section so the operating system can schedule

[5] These are typical run-times. When using busy-waiting and the number of threads is greater than the number of cores, the run-times vary considerably.

Table 4.1 Run-times (in seconds) of π programs using $n = 10^8$ terms on a system with two four-core processors.

Threads	Busy-Wait	Mutex
1	2.90	2.90
2	1.45	1.45
4	0.73	0.73
8	0.38	0.38
16	0.50	0.38
32	0.80	0.40
64	3.56	0.38

Table 4.2 Possible sequence of events with busy-waiting and more threads than cores.

		Thread				
Time	`flag`	**0**	**1**	**2**	**3**	**4**
0	0	crit sect	busy wait	susp	susp	susp
1	1	terminate	crit sect	susp	busy wait	susp
2	2	—	terminate	susp	busy wait	busy wait
⋮	⋮			⋮	⋮	⋮
?	2	—	—	crit sect	susp	busy wait

thread 2, thread 3, or thread 4. Suppose it schedules thread 3, which will spin in the **while** loop. When thread 1 finishes the critical section and sets `flag` = 2, the operating system can schedule thread 2 or thread 4. If it schedules thread 4, then both thread 3 and thread 4 will be busily spinning in the busy-wait loop until the operating system deschedules one of them and schedules thread 2. (See Table 4.2.)

4.7 Producer–consumer synchronization and semaphores

Although busy-waiting is generally wasteful of CPU resources, it does have the property that we know, in advance, the order in which the threads will execute the code in the critical section: thread 0 is first, then thread 1, then thread 2, and so on. With mutexes, the order in which the threads execute the critical section is left to chance and the system. Since addition is commutative, this doesn't matter in our program for estimating π. However, it's not difficult to think of situations in which we also want to control the order in which the threads execute the code in the critical section. For example, suppose each thread generates an $n \times n$ matrix, and we want to multiply the matrices together in thread-rank order. Since matrix multiplication isn't commutative, our mutex solution would have problems:

```
/* n and product_matrix are shared and initialized by
 * the main thread. product_matrix is initialized
 * to be the identity matrix. */
void* Thread_work(void* rank) {
   long my_rank = (long) rank;
   matrix_t my_mat = Allocate_matrix(n);
   Generate_matrix(my_mat);
   pthread_mutex_lock(&mutex);
   Multiply_matrix(product_mat, my_mat);
   pthread_mutex_unlock(&mutex);
   Free_matrix(&my_mat);
   return NULL;
}  /* Thread_work */
```

A somewhat more complicated example involves having each thread "send a message" to another thread. For example, suppose we have `thread_count` or t threads and we want thread 0 to send a message to thread 1, thread 1 to send a message to thread 2, ..., thread $t-2$ to send a message to thread $t-1$ and thread $t-1$ to send a message to thread 0. After a thread "receives" a message, it can print the message and terminate. To implement the message transfer, we can allocate a shared array of **char***. Then each thread can allocate storage for the message it's sending, and, after it has initialized the message, set a pointer in the shared array to refer to it. To avoid dereferencing undefined pointers, the main thread can set the individual entries in the shared array to `NULL`. (See Program 4.7.) When we run the program with more

```
/* 'messages' has type char**. It's allocated in main. */
/* Each entry is set to NULL in main.                   */
void *Send_msg(void* rank) {
   long my_rank = (long) rank;
   long dest = (my_rank + 1) % thread_count;
   long source = (my_rank + thread_count - 1) % thread_count;
   char* my_msg = malloc(MSG_MAX*sizeof(char));

   sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
   messages[dest] = my_msg;

   if (messages[my_rank] != NULL)
      printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
   else
      printf("Thread %ld > No message from %ld\n",
            my_rank, source);

   return NULL;
}  /* Send_msg */
```

Program 4.7: A first attempt at sending messages using pthreads.

than a couple of threads on a dual core system, we see that some of the messages are never received. For example, thread 0, which is started first, will typically finish before thread $t - 1$ has copied the message into the `messages` array.

This isn't surprising, and we could fix the problem by replacing the **if** statement in Line 12 with a busy-wait **while** statement:

```
while (messages[my_rank] == NULL);
printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
```

Of course, this solution would have the same problems that any busy-waiting solution has, so we'd prefer a different approach.

After executing the assignment in Line 10, we'd like to "notify" the thread with rank `dest` that it can proceed to print the message. We'd like to do something like this:

```
. . .
messages[dest] = my_msg;
Notify thread dest that it can proceed;

Await notification from thread source
printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
. . .
```

It's not at all clear how mutexes can help here. We might try calling `pthread_mutex_unlock` to "notify" the thread `dest`. However, mutexes are initialized to be *unlocked*, so we'd need to add a call *before* initializing `messages[dest]` to lock the mutex. This will be a problem, since we don't know when the threads will reach the calls to `pthread_mutex_lock`.

To make this a little clearer, suppose that the main thread creates and initializes an array of mutexes, one for each thread. Then we're trying to do something like this:

```
1    . . .
2    pthread_mutex_lock(&mutex[dest]);
3    . . .
4    messages[dest] = my_msg;
5    pthread_mutex_unlock(&mutex[dest]);
6    . . .
7    pthread_mutex_lock(&mutex[my_rank]);
8    printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
9    . . .
```

Now suppose we have two threads, and thread 0 gets so far ahead of thread 1 that it reaches the second call to `pthread_mutex_lock` in Line 7 before thread 1 reaches the first in Line 2. Then, of course, it will acquire the lock and continue to the `printf` statement. This will result in thread 0 dereferencing a null pointer and it will crash.

There *are* other approaches to solving this problem with mutexes. See, for example, Exercise 4.8. However, POSIX also provides a somewhat different means of controlling access to critical sections: **semaphores**. Let's take a look at them.

A semaphore can be thought of as a special type of **unsigned int**, so they take on the values 0, 1, 2, ... In many cases, we'll only be interested in using them when they take on the values 0 and 1. A semaphore that only takes on these values is called a *binary* semaphore. Very roughly speaking, 0 corresponds to a locked mutex, and 1 corresponds to an unlocked mutex. To use a binary semaphore as a mutex, you *initialize* it to 1: "unlocked." Before the critical section you want to protect, you place a call to the function `sem_wait`. A thread that executes `sem_wait` will block if the semaphore is 0. If the semaphore is nonzero, it will *decrement* the semaphore and proceed. After executing the code in the critical section, a thread calls `sem_post`, which *increments* the semaphore, and a thread waiting in `sem_wait` can proceed.

Semaphores were first defined by the computer scientist Edsger Dijkstra in [15]. The name is taken from the mechanical device that railroads use to control which train can use a track. The device consists of an arm attached by a pivot to a post. When the arm points down, approaching trains can proceed, and when the arm is perpendicular to the post, approaching trains must stop and wait. The track corresponds to the critical section: when the arm is down corresponds to a semaphore of 1, and when the arm is up corresponds to a semaphore of 0. The `sem_wait` and `sem_post` calls correspond to signals sent by the train to the semaphore controller.

For our current purposes, the crucial difference between semaphores and mutexes is that there is no ownership associated with a semaphore. The main thread can initialize all of the semaphores to 0—that is, "locked"—and then any thread can execute a `sem_post` on any of the semaphores. Similarly, any thread can execute `sem_wait` on any of the semaphores. Thus, if we use semaphores, our `Send_msg` function can be written as shown in Program 4.8.

The syntax of the various semaphore functions is

```
int sem_init(
      sem_t*      semaphore_p    /* out */,
      int         shared         /* in  */,
      unsigned    initial_val    /* in  */);

int sem_destroy(sem_t*   semaphore_p   /* in/out */);
int sem_post(sem_t*      semaphore_p   /* in/out */);
int sem_wait(sem_t*      semaphore_p   /* in/out */);
```

The second argument to `sem_init` controls whether the semaphore is shared among threads or processes. In our examples, we'll be sharing the semaphore among threads, so the constant 0 can be passed in.

Note that semaphores are part of the POSIX standard, but *not* part of Pthreads. Hence it is necessary to ensure your operating system does indeed support semaphores, and then add the following preprocessor directive to any program that uses them[6]:

[6] Some systems, including macOS, don't support this version of semaphores. However, they may support something called "named" semaphores. The functions `sem_wait` and `sem_post` can be used in the same

```
/* 'messages' is allocated and initialized to NULL in main */
/* 'semaphores' is allocated and initialized to            */
/* 0 (locked) in main                                      */
void *Send_msg(void* rank) {
   long my_rank = (long) rank;
   long dest = (my_rank + 1) % thread_count;
   char* my_msg = malloc(MSG_MAX*sizeof(char));

   sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
   messages[dest] = my_msg;
   /* ``Unlock'' the semaphore of dest: */
   sem_post(&semaphores[dest]);

   /* Wait for our semaphore to be unlocked */
   sem_wait(&semaphores[my_rank]);
   printf("Thread %ld > %s\n", my_rank, messages[my_rank]);

   return NULL;
}  /* Send_msg */
```

Program 4.8: Using semaphores so that threads can send messages.

```
#include <semaphore.h>
```

Finally, note that the message-sending problem didn't involve a critical section. The problem wasn't that there was a block of code that could only be executed by one thread at a time. Rather, thread `my_rank` couldn't proceed until thread `source` had finished creating the message. This type of synchronization, when a thread can't proceed until another thread has taken some action, is sometimes called **producer–consumer synchronization**. For example, imagine a *producer* thread that generates tasks and places them in a fixed-size queue (or *bounded buffer*) for a *consumer* thread to execute. In this case, the consumer blocks until at least one task is ready, at which point it will be signaled by the producer. Once signaled, the work is carried out by the thread in isolation; no critical section is involved. This paradigm is seen in stream processing, web servers, and so on; in the case of a web server, the producer thread could listen for incoming request URIs and place them in the queue, while the consumer would be responsible for reading the corresponding file from disk (e.g., `http://server/file.txt` might be located at `/www/file.txt` on the web server's file system) and sending data back to the client that requested the URI.

As mentioned earlier, binary semaphores (those that only take on the values 0 and 1) are fairly typical. However, *counting* semaphores can also be useful in scenar-

way. However, `sem_init` should be replaced by `sem_open`, and `sem_destroy` should be replaced by `sem_close` and `sem_unlink`. See the book's website for an example.

ios where we wish to restrict access to a finite resource. One common example is an application design pattern that involves limiting the number of threads used by a program to be no more than the number of cores available on a given machine. Consider a program with a workload of N tasks, where N is much greater than the available cores. In this case, the main thread is responsible for distributing the workload and would initialize its semaphore with the number of cores available, and then call `sem_wait` before starting each worker thread with `pthread_create`. Once the counter reaches 0, the main thread will block; the machine has a task running for each core and the program must wait for a thread to finish before starting more. When a thread does finish its task, it will call `sem_post` to signal that the main thread can create another worker thread. For this approach to be efficient, the amount of time spent on each task much be longer than the thread creation overhead because N total threads will be started during the program's execution. For an approach that reuses existing threads in a *thread pool*, see Programming Assignment 4.5.

4.8 Barriers and condition variables

Let's take a look at another problem in shared-memory programming: synchronizing the threads by making sure that they all are at the same point in a program. Such a point of synchronization is called a **barrier**, because no thread can proceed beyond the barrier until all the threads have reached it.

Barriers have numerous applications. As we discussed in Chapter 2, if we're timing some part of a multithreaded program, we'd like for all the threads to start the timed code at the same instant, and then report the time taken by the last thread to finish, i.e., the "slowest" thread. So we'd like to do something like this:

```
/* Shared */
double elapsed_time;
. . .
/* Private */
double my_start, my_finish, my_elapsed;
. . .
Synchronize threads;
Store current time in my_start;
/* Execute timed code */
. . .
Store current time in my_finish;
my_elapsed = my_finish − my_start;

elapsed = Maximum of my_elapsed values;
```

Using this approach, we're sure that all of the threads will record `my_start` at approximately the same time.

Another very important use of barriers is in debugging. As you've probably already seen, it can be very difficult to determine *where* an error is occurring in a parallel program. We can, of course, have each thread print a message indicating which point it's reached in the program, but it doesn't take long for the volume of the output to become overwhelming. Barriers provide an alternative:

```
point in program we want to reach;
barrier;
if (my_rank == 0) {
   printf("All threads reached this point\n");
   fflush(stdout);
}
```

Many implementations of Pthreads don't provide barriers, so if our code is to be portable, we need to develop our own implementation. There are a number of options; we'll look at three. The first two only use constructs that we've already studied. The third uses a new type of Pthreads object: a *condition variable*.

4.8.1 Busy-waiting and a mutex

Implementing a barrier using busy-waiting and a mutex is straightforward: we use a shared counter protected by the mutex. When the counter indicates that every thread has entered the critical section, threads can leave the busy-wait loop.

```
/* Shared and initialized by the main thread */
int counter;  /* Initialize to 0 */
int thread_count;
pthread_mutex_t barrier_mutex;
. . .

void* Thread_work(. . .) {
   . . .
   /* Barrier */
   pthread_mutex_lock(&barrier_mutex);
   counter++;
   pthread_mutex_unlock(&barrier_mutex);
   while (counter < thread_count);
   . . .
}
```

Of course, this implementation will have the same problems that our other busy-wait codes had: we'll waste CPU cycles when threads are in the busy-wait loop, and, if we run the program with more threads than cores, we may find that the performance of the program seriously degrades.

Another issue is the shared variable `counter`. What happens if we want to implement a second barrier and we try to reuse the counter? When the first barrier is completed, `counter` will have the value `thread_count`. Unless we can somehow reset `counter`, the **while** condition we used for our first barrier `counter < thread_count` will be false, and the barrier won't cause the threads to block. Furthermore, any attempt to reset `counter` to zero is almost certainly doomed to failure. If the last thread to enter the loop tries to reset it, some thread in the busy-wait may never see the fact that counter == thread_count, and that thread may hang in the busy-wait. If some thread tries to reset the counter after the barrier, some other thread may enter the second barrier before the counter is reset and its increment to the counter will be lost. This will have the unfortunate effect of causing all the threads to hang in the second busy-wait loop. So if we want to use this barrier, we need one counter variable for each instance of the barrier.

4.8.2 Semaphores

A natural question is whether we can implement a barrier with semaphores, and, if so, whether we can reduce the number of problems we encountered with busy-waiting. The answer to the first question is yes:

```
/* Shared variables */
int counter;          /* Initialize to 0 */
sem_t count_sem;      /* Initialize to 1 */
sem_t barrier_sem;    /* Initialize to 0 */
. . .
void* Thread_work(...) {
   . . .
   /* Barrier */
   sem_wait(&count_sem);
   if (counter == thread_count-1) {
      counter = 0;
      sem_post(&count_sem);
      for (j = 0; j < thread_count-1; j++)
         sem_post(&barrier_sem);
   } else {
      counter++;
      sem_post(&count_sem);
      sem_wait(&barrier_sem);
   }
   . . .
}
```

As with the busy-wait barrier, we have a counter that we use to determine how many threads have entered the barrier. We use two semaphores: count_sem protects the counter, and barrier_sem is used to block threads that have entered the barrier. The count_sem semaphore is initialized to 1 (that is, “unlocked”), so the first thread to reach the barrier will be able to proceed past the call to sem_wait. Subsequent threads, however, will block until they can have exclusive access to the counter. When a thread has exclusive access to the counter, it checks to see if counter < thread_count-1. If it is, the thread increments counter, relinquishes the lock (sem_post(&count_sem)), and blocks in sem_wait(&barrier_sem). On the other hand, if counter == thread_count-1, the thread is the last to enter the barrier, so it can reset `counter` to zero and "unlock" `count_sem` by calling `sem_post(&count_sem)`. Now, it wants to notify all the other threads that they can proceed, so it executes `sem_post(&barrier_sem)` for each of the `thread_count`-1 threads that are blocked in `sem_wait(&barrier_sem)`.

Note that it doesn't matter if the thread executing the loop of calls to `sem_post(&barrier_sem)` races ahead and executes multiple calls to `sem_post` before a thread can be unblocked from `sem_wait(&barrier_sem)`. Recall that a semaphore is an **unsigned int**, and the calls to `sem_post` increment it, while the calls to `sem_wait` decrement it—unless it's already 0. If it's 0, the calling threads will block until it's positive again. Therefore it doesn't matter if the thread executing the loop of calls to `sem_post(&barrier_sem)` gets ahead of the threads blocked in the calls to `sem_wait(&barrier_sem)`, because eventually the blocked threads will see that `barrier_sem` is positive, and they'll decrement it and proceed.

It should be clear that this implementation of a barrier is superior to the busy-wait barrier, since the threads don't need to consume CPU cycles when they're blocked in `sem_wait`. Can we reuse the data structures from the first barrier if we want to execute a second barrier?

The `counter` can be reused, since we were careful to reset it before releasing any of the threads from the barrier. Also, `count_sem` can be reused, since it is reset to 1 before any threads can leave the barrier. This leaves `barrier_sem`. Since there's exactly one `sem_post` for each `sem_wait`, it might appear that the value of `barrier_sem` will be 0 when the threads start executing a second barrier. However, suppose we have two threads, and thread 0 is blocked in `sem_wait(&barrier_sem)` in the first barrier, while thread 1 is executing the loop of calls to `sem_post`. Also suppose that the operating system has seen that thread 0 is idle, and descheduled it out. Then thread 1 can go on to the second barrier. Since `counter` == 0, it will execute the **else** clause. After incrementing `counter`, it executes `sem_post(&count_sem)`, and then executes `sem_wait(&barrier_sem)`.

However, if thread 0 is still descheduled, it will not have decremented `barrier_sem`. Thus when thread 1 reaches `sem_wait(&barrier_sem)`, `barrier_sem` will still be 1, so it will simply decrement `barrier_sem` and proceed. This will have the unfortunate consequence that when thread 0 starts executing again, it will still be blocked in the *first* `sem_wait(&barrier_sem)`, and thread 1 will proceed through the second barrier before thread 0 has entered it. Reusing `barrier_sem` therefore results in a race condition.

4.8.3 Condition variables

A somewhat better approach to creating a barrier in Pthreads is provided by *condition variables*. A **condition variable** is a data object that allows a thread to suspend execution until a certain event or *condition* occurs. When the event or condition occurs another thread can *signal* the thread to "wake up." A condition variable is *always* associated with a mutex.

Typically, condition variables are used in constructs similar to this pseudocode:

```
lock mutex;
if condition has occurred
   signal thread(s);
else {
   unlock the mutex and block;
   /* when thread is unblocked, mutex is relocked */
}
unlock mutex;
```

Condition variables in Pthreads have type `pthread_cond_t`. The function

```
int pthread_cond_signal(
   pthread_cond_t*  cond_var_p  /* in/out */);
```

will unblock *one* of the blocked threads, and

```
int pthread_cond_broadcast(
   pthread_cond_t* cond_var_p  /* in/out */);
```

will unblock *all* of the blocked threads. This is one advantage of condition variables; recall that we needed a **for** loop calling `sem_post` to achieve similar functionality with semaphores. The function

```
int pthread_cond_wait(
      pthread_cond_t*    cond_var_p   /* in/out */,
      pthread_mutex_t*   mutex_p      /* in/out */);
```

will unlock the mutex referred to by `mutex_p` and cause the executing thread to block until it is unblocked by another thread's call to `pthread_cond_signal` or `pthread_cond_broadcast`. When the thread is unblocked, it reacquires the mutex. So in effect, `pthread_cond_wait` implements the following sequence of functions:

```
pthread_mutex_unlock(&mutex_p);
wait_on_signal(&cond_var_p);
pthread_mutex_lock(&mutex_p);
```

The following code implements a barrier with a condition variable:

```
/* Shared */
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_t cond_var;
. . .
void* Thread_work(. . .) {
   . . .
   /* Barrier */
   pthread_mutex_lock(&mutex);
   counter++;
   if (counter == thread_count) {
      counter = 0;
      pthread_cond_broadcast(&cond_var);
   } else {
         while (pthread_cond_wait(&cond_var, &mutex) != 0);
      }
      pthread_mutex_unlock(&mutex);
      . . .
   }
```

Note that it is possible that events other than the call to pthread_cond_broadcast can cause a suspended thread to unblock (see, for example, Butenhof [7], page 80). This is called a *spurious wake-up*. Hence, the call to pthread_cond_wait should usually be placed in a **while** loop. If the thread is unblocked by some event other than a call to pthread_cond_signal or pthread_cond_broadcast, then the return value of pthread_cond_wait will be nonzero, and the unblocked thread will call pthread_cond_wait again.

If a single thread is being awakened, it's also a good idea to check that the condition has, in fact, been satisfied before proceeding. In our example, if a single thread were being released from the barrier with a call to pthread_cond_signal, then that thread should verify that counter == 0 before proceeding. This can be dangerous with the broadcast, though. After being awakened, some thread may race ahead and change the condition, and if each thread is checking the condition, a thread that awakened later may find the condition is no longer satisfied and go back to sleep.

Note that in order for our barrier to function correctly, it's essential that the call to pthread_cond_wait unlock the mutex. If it didn't unlock the mutex, then only one thread could enter the barrier; all of the other threads would block in the call to pthread_mutex_lock, the first thread to enter the barrier would block in the call to pthread_cond_wait, and our program would hang.

Also note that the semantics of mutexes require that the mutex be relocked before we return from the call to pthread_cond_wait. We obtained the lock when we returned from the call to pthread_mutex_lock. Hence, we should at some point relinquish the lock through a call to pthread_mutex_unlock.

Like mutexes and semaphores, condition variables should be initialized and destroyed. In this case, the functions are

```
int pthread_cond_init(
      pthread_cond_t*            cond_p      /* out */,
      const pthread_condattr_t*  cond_attr_p /* in  */);

int pthread_cond_destroy(
   pthread_cond_t*  cond_p  /* in/out */);
```

We won't be using the second argument to pthread_cond_init—as with mutexes, the default the attributes are fine for our purposes—so we'll call it with second argument set to NULL. As usual, there is also a *static* version of the initializer if we are planning to use the default attributes:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Condition variables are often quite useful whenever a thread needs to wait for something. When protected application state cannot be represented by an unsigned integer counter, condition variables may be preferable to semaphores.

4.8.4 Pthreads barriers

Before proceeding we should note that the Open Group, the standards group that is continuing to develop the POSIX standard, does define a barrier interface for Pthreads. However, as we noted earlier, it is not universally available, so we haven't discussed it in the text. See Exercise 4.10 for some of the details of the API.

4.9 Read-write locks

Let's take a look at the problem of controlling access to a large, shared data structure, which can be either simply searched or updated by the threads. For the sake of explicitness, let's suppose the shared data structure is a sorted, singly-linked list of **int**s, and the operations of interest are `Member`, `Insert`, and `Delete`.

4.9.1 Sorted linked list functions

The list itself is composed of a collection of list *nodes*, each of which is a struct with two members: an **int** and a pointer to the next node. We can define such a struct with the definition

```
struct list_node_s {
   int data;
   struct list_node_s* next;
}
```

A typical list is shown in Fig. 4.4. A pointer, `head_p`, with type **struct** `list_node_s*` refers to the first node in the list. The `next` member of the last node is `NULL` (which is indicated by a slash (/) in the `next` member).

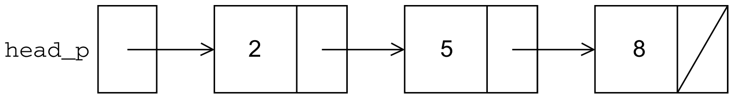

FIGURE 4.4

A linked list.

The `Member` function (Program 4.9) uses a pointer to traverse the list until it either finds the desired value or determines that the desired value cannot be in the list. Since the list is sorted, the latter condition occurs when the `curr_p` pointer is `NULL` or when the data member of the current node is larger than the desired value.

The `Insert` function (Program 4.10) begins by searching for the correct position in which to insert the new node. Since the list is sorted, it must search until it finds a node whose `data` member is greater than the `value` to be inserted. When it finds this node, it needs to insert the new node in the position *preceding* the node that's been found. Since the list is singly-linked, we can't "back up" to this position without traversing the list a second time. There are several approaches to dealing with this;

```
int  Member(int value, struct list_node_s* head_p) {
   struct list_node_s* curr_p = head_p;

   while (curr_p != NULL && curr_p->data < value)
      curr_p = curr_p->next;

   if (curr_p == NULL || curr_p->data > value) {
      return 0;
   } else {
      return 1;
   }
}  /* Member */
```

Program 4.9: The `Member` function.

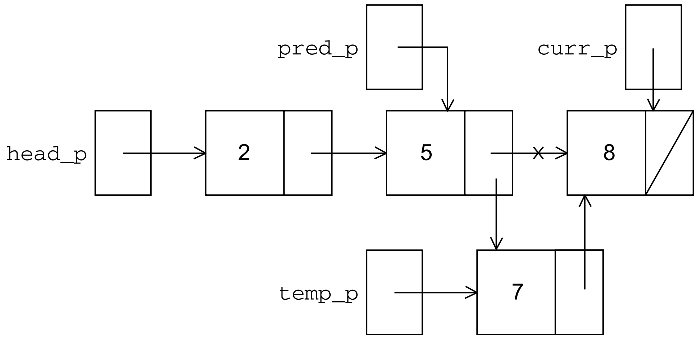

FIGURE 4.5

Inserting a new node into a list.

the approach we use is to define a second pointer `pred_p`, which, in general, refers to the predecessor of the current node. When we exit the loop that searches for the position to insert, the `next` member of the node referred to by `pred_p` can be updated so that it refers to the new node. (See Fig. 4.5.)

The `Delete` function (Program 4.11) is similar to the `Insert` function in that it also needs to keep track of the predecessor of the current node while it's searching for the node to be deleted. The predecessor node's `next` member can then be updated after the search is completed. (See Fig. 4.6.)

4.9.2 A multithreaded linked list

Now let's try to use these functions in a Pthreads program. To share access to the list, we can define `head_p` to be a global variable. This will simplify the function headers for `Member`, `Insert`, and `Delete`, since we won't need to pass in either `head_p` or a

```
int Insert(int value, struct list_node_s** head_pp) {
   struct list_node_s* curr_p = *head_pp;
   struct list_node_s* pred_p = NULL;
   struct list_node_s* temp_p;

   while (curr_p != NULL && curr_p->data < value) {
      pred_p = curr_p;
      curr_p = curr_p->next;
   }

   if (curr_p == NULL || curr_p->data > value) {
      temp_p = malloc(sizeof(struct list_node_s));
      temp_p->data = value;
      temp_p->next = curr_p;
      if (pred_p == NULL) /* New first node */
         *head_pp = temp_p;
      else
         pred_p->next = temp_p;
      return 1;
   } else { /* Value already in list */
      return 0;
   }
} /* Insert */
```

Program 4.10: The `Insert` function.

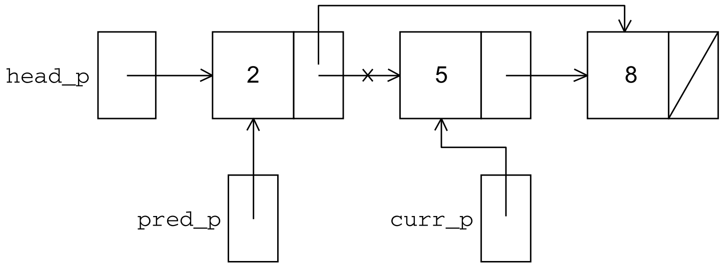

FIGURE 4.6

Deleting a node from the list.

pointer to `head_p`, we'll only need to pass in the value of interest. What now are the consequences of having multiple threads simultaneously execute the three functions?

Since multiple threads can simultaneously *read* a memory location without conflict, it should be clear that multiple threads can simultaneously execute `Member`. On the other hand, `Delete` and `Insert` also *write* to memory locations, so there may be problems if we try to execute either of these operations at the same time as another

```
int Delete(int value, struct list_node_s** head_pp) {
   struct list_node_s* curr_p = *head_pp;
   struct list_node_s* pred_p = NULL;

   while (curr_p != NULL && curr_p->data < value) {
      pred_p = curr_p;
      curr_p = curr_p->next;
   }

   if (curr_p != NULL && curr_p->data == value) {
      if (pred_p == NULL) { /* Deleting first node in list */
         *head_pp = curr_p->next;
         free(curr_p);
      } else {
         pred_p->next = curr_p->next;
         free(curr_p);
      }
      return 1;
   } else { /* Value isn't in list */
      return 0;
   }
}  /* Delete */
```

Program 4.11: The `Delete` function.

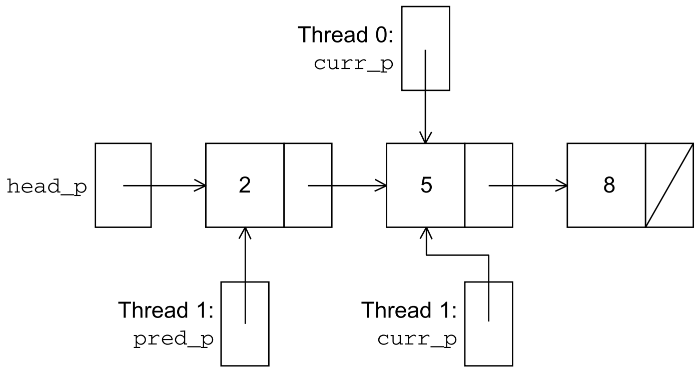

FIGURE 4.7

Simultaneous access by two threads.

operation. As an example, suppose that thread 0 is executing `Member`(5) at the same time that thread 1 is executing `Delete`(5), and the current state of the list is shown in Fig. 4.7. An obvious problem is that if thread 0 is executing `Member`(5), it is going to report that 5 is in the list, when, in fact, it may be deleted even before thread 0 returns.

A second obvious problem is if thread 0 is executing `Member(8)`, thread 1 may free the memory used for the node storing 5 before thread 0 can advance to the node storing 8. Although typical implementations of `free` don't overwrite the freed memory, if the memory is reallocated before thread 0 advances, there can be serious problems. For example, if the memory is reallocated for use in something other than a list node, what thread 0 "thinks" is the `next` member may be set to utter garbage, and after it executes

```
curr_p = curr_p->next;
```

dereferencing `curr_p` may result in a segmentation violation.

More generally, we can run into problems if we try to simultaneously execute another operation while we're executing an `Insert` or a `Delete`. It's OK for multiple threads to simultaneously execute `Member`—that is, *read* the list nodes—but it's unsafe for multiple threads to access the list if at least one of the threads is executing an `Insert` or a `Delete`—that is, is *writing* to the list nodes (see Exercise 4.12).

How can we deal with this problem? An obvious solution is to simply lock the list any time that a thread attempts to access it. For example, a call to each of the three functions can be protected by a mutex, so we might execute

```
Pthread_mutex_lock(&list_mutex);
Member(value);
Pthread_mutex_unlock(&list_mutex);
```

instead of simply calling `Member(value)`.

An equally obvious problem with this solution is that we are serializing access to the list, and if the vast majority of our operations are calls to `Member`, we'll fail to exploit this opportunity for parallelism. On the other hand, if most of our operations are calls to `Insert` and `Delete`, then this may be the best solution, since we'll need to serialize access to the list for most of the operations, and this solution will certainly be easy to implement.

An alternative to this approach involves "finer-grained" locking. Instead of locking the entire list, we could try to lock individual nodes. We would add, for example, a mutex to the list node struct:

```
struct list_node_s {
   int data;
   struct list_node_s* next;
   pthread_mutex_t mutex;
}
```

Now each time we try to access a node we must first lock the mutex associated with the node. Note that this will also require that we have a mutex associated with the `head_p` pointer. So, for example, we might implement `Member` as shown in Program 4.12. Admittedly this implementation is *much* more complex than the original `Member` function. It is also much slower, since, in general, each time a node is accessed, a mutex must be locked and unlocked. At a minimum it will add two function calls to the node access, but it can also add a substantial delay if a thread has to wait

```
int  Member(int value) {
   struct list_node_s* temp_p;

   pthread_mutex_lock(&head_p_mutex);
   temp_p = head_p;
   while (temp_p != NULL && temp_p->data < value) {
      if (temp_p->next != NULL)
         pthread_mutex_lock(&(temp_p->next->mutex));
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      pthread_mutex_unlock(&(temp_p->mutex));
      temp_p = temp_p->next;
   }

   if (temp_p == NULL || temp_p->data > value) {
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      if (temp_p != NULL)
         pthread_mutex_unlock(&(temp_p->mutex));
      return 0;
   } else {
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      pthread_mutex_unlock(&(temp_p->mutex));
      return 1;
   }
}  /* Member */
```

Program 4.12: Implementation of `Member` with one mutex per list node.

for a lock. A further problem is that the addition of a mutex field to each node will substantially increase the amount of storage needed for the list. On the other hand, the finer-grained locking might be a closer approximation to what we want. Since we're only locking the nodes of current interest, multiple threads can simultaneously access different parts of the list, regardless of which operations they're executing.

4.9.3 Pthreads read-write locks

Neither of our multithreaded linked lists exploits the potential for simultaneous access to *any* node by threads that are executing `Member`. The first solution only allows one thread to access the entire list at any instant, and the second only allows one thread to access any given node at any instant. An alternative is provided by Pthreads' **read-write locks**. A read-write lock is somewhat like a mutex except that it provides *two* lock functions. The first lock function locks the read-write lock for *reading*, while the second locks it for *writing*. Multiple threads can thereby simultaneously obtain the lock by calling the read-lock function, while only one thread can obtain the lock by calling the write-lock function. Thus if any threads own the lock for reading, any threads that want to obtain the lock for writing will block in the call to the write-lock function. Furthermore, if any thread owns the lock for writing, any threads that want to obtain the lock for reading or writing will block in their respective locking functions.

Using Pthreads read-write locks, we can protect our linked list functions with the following code (we're ignoring function return values):

```
pthread_rwlock_rdlock(&rwlock);
Member(value);
pthread_rwlock_unlock(&rwlock);
. . .
pthread_rwlock_wrlock(&rwlock);
Insert(value);
pthread_rwlock_unlock(&rwlock);
. . .
pthread_rwlock_wrlock(&rwlock);
Delete(value);
pthread_rwlock_unlock(&rwlock);
```

The syntax for the new Pthreads functions is

```
int pthread_rwlock_rdlock(
   pthread_rwlock_t*  rwlock_p  /* in/out */);

int pthread_rwlock_wrlock(
   pthread_rwlock_t*  rwlock_p  /* in/out */);

int pthread_rwlock_unlock(
   pthread_rwlock_t*  rwlock_p  /* in/out */);
```

As their names suggest, the first function locks the read-write lock for reading, the second locks it for writing, and the last unlocks it.

As with mutexes, read-write locks should be initialized before use and destroyed after use. The following function can be used for initialization:

```
int pthread_rwlock_init(
       pthread_rwlock_t*            rwlock_p  /* out */,
       const pthread_rwlockattr_t*  attr_p    /* in  */);

/* And, the static version: */
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
```

Also as with mutexes, we'll not use the second argument, so we'll just pass `NULL`. The following function can be used for destruction of a read-write lock:

```
int pthread_rwlock_destroy(
   pthread_rwlock_t*  rwlock_p  /* in/out*/);
```

4.9.4 Performance of the various implementations

Of course, we really want to know which of the three implementations is "best," so we included our implementations in a small program in which the main thread first inserts a user-specified number of randomly generated keys into an empty list. After being started by the main thread, each thread carries out a user-specified number of operations on the list. The user also specifies the percentages of each type of operation (`Member`, `Insert`, `Delete`). However, which operation occurs when and on which key is determined by a random number generator. Thus, for example, the user might specify that 1000 keys should be inserted into an initially empty list and a total of 100,000 operations are to be carried out by the threads. Further, she might specify that 80% of the operations should be `Member`, 15% should be `Insert`, and the remaining 5% should be `Delete`. However, since the operations are randomly generated, it might happen that the threads execute a total of, say, 79,000 calls to `Member`, 15,500 calls to `Insert`, and 5500 calls to `Delete`.

Tables 4.3 and 4.4 show the times (in seconds) that it took for 100,000 operations on a list that was initialized to contain 1000 keys. Both sets of data were taken on a system containing four dual-core processors.

Table 4.3 Linked list times: 100,000 ops/thread, 99.9% `Member`, 0.05% `Insert`, 0.05% `Delete`.

	Number of Threads			
Implementation	**1**	**2**	**4**	**8**
Read-Write Locks	0.213	0.123	0.098	0.115
One Mutex for Entire List	0.211	0.450	0.385	0.457
One Mutex per Node	1.680	5.700	3.450	2.700

Table 4.4 Linked list times: 100,000 ops/thread, 80% `Member`, 10% `Insert`, 10% `Delete`.

	Number of Threads			
Implementation	**1**	**2**	**4**	**8**
Read-Write Locks	2.48	4.97	4.69	4.71
One Mutex for Entire List	2.50	5.13	5.04	5.11
One Mutex per Node	12.00	29.60	17.00	12.00

Table 4.3 shows the times when 99.9% of the operations are `Member` and the remaining 0.1% are divided equally between `Insert` and `Delete`. Table 4.4 shows the times when 80% of the operations are `Member`, 10% are `Insert`, and 10% are Delete. Note that in both tables when one thread is used, the run-times for the read-write locks and the single-mutex implementations are about the same. This makes sense: the operations are serialized, and since there is no contention for the read-write lock or the mutex, the overhead associated with both implementations should consist of a function call before the list operation and a function call after the operation. On the

other hand, the implementation that uses one mutex per node is *much* slower. This also makes sense, since each time a single node is accessed, there will be two function calls—one to lock the node mutex and one to unlock it. Thus there's considerably more overhead for this implementation.

The inferiority of the implementation that uses one mutex per node persists when we use multiple threads. There is far too much overhead associated with all the locking and unlocking to make this implementation competitive with the other two implementations.

Perhaps the most striking difference between the two tables is the relative performance of the read-write lock implementation and the single-mutex implementation when multiple threads are used. When there are very few `Insert`s and `Delete`s, the read-write lock implementation is far better than the single-mutex implementation. Since the single-mutex implementation will serialize all the operations, this suggests that if there are very few `Insert`s and `Delete`s, the read-write locks do a very good job of allowing concurrent access to the list. On the other hand, if there are a relatively large number of `Insert`s and `Delete`s (for example, 10% each), there's very little difference between the performance of the read-write lock implementation and the single-mutex implementation. Thus, for linked list operations, read-write locks *can* provide a considerable increase in performance, but only if the number of `Insert`s and `Delete`s is quite small.

Also notice that if we use one mutex or one mutex per node, the program is *always* as fast or faster when it's run with one thread. Furthermore, when the number of `Insert`s and `Delete`s is relatively large, the read-write lock program is also faster with one thread. This isn't surprising for the one mutex implementation, since effectively accesses to the list are serialized. For the read-write lock implementation, it appears that when there are a substantial number of write locks, there is too much contention for the locks and overall performance deteriorates significantly.

In summary, the read-write lock implementation is superior to the single mutex and one mutex per node implementations. However, unless the number of `Insert`s and `Delete`s is small, a serial implementation will be superior.

4.9.5 Implementing read-write locks

The original Pthreads specification didn't include read-write locks, so some of the early texts describing Pthreads include implementations of read-write locks (see, for example, [7]). A typical implementation[7] defines a data structure that uses two condition variables—one for "readers" and one for "writers"—and a mutex. The structure also contains members that indicate

1. how many readers own the lock, that is, are currently reading,
2. how many readers are waiting to obtain the lock,

[7] This discussion follows the basic outline of Butenhof's implementation [7].

3. whether a writer owns the lock, and
4. how many writers are waiting to obtain the lock.

The mutex protects the read-write lock data structure: whenever a thread calls one of the functions (read-lock, write-lock, unlock), it first locks the mutex, and whenever a thread completes one of these calls, it unlocks the mutex. After acquiring the mutex, the thread checks the appropriate data members to determine how to proceed. As an example, if it wants read-access, it can check to see if there's a writer that currently owns the lock. If not, it increments the number of active readers and proceeds. If a writer is active, it increments the number of readers waiting and starts a condition wait on the reader condition variable. When it's awakened, it decrements the number of readers waiting, increments the number of active readers, and proceeds. The write-lock function has an implementation that's similar to the read-lock function.

The action taken in the unlock function depends on whether the thread was a reader or a writer. If the thread was a reader, there are no currently active readers, *and* there's a writer waiting, then it can signal a writer to proceed before returning. If, on the other hand, the thread was a writer, there can be both readers and writers waiting, so the thread needs to decide whether it will give preference to readers or writers. Since writers must have exclusive access, it is likely that it is much more difficult for a writer to obtain the lock. Many implementations therefore give writers preference. Programming Assignment 4.6 explores this further.

4.10 Caches, cache-coherence, and false sharing[8]

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

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

In Section 2.3.5, we saw that the use of cache memory can have a huge impact on shared memory. Let's recall why. First, consider the following situation: Suppose

[8] This material is also covered in Chapter 5. So if you've already read that chapter, you may want to skim this section.

x is a shared variable with the value five, and both thread 0 and thread 1 read x from memory into their (separate) caches, because both want to execute the statement

```
my_y = x;
```

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

```
x++;
```

Finally, suppose that thread 1 now executes

```
my_z = x;
```

where my_z is another private variable.

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

The use of cache coherence can have a dramatic effect on the performance of shared-memory systems. To illustrate this, recall our Pthreads matrix-vector multiplication example: the main thread initialized an $m \times n$ matrix A and an n-dimensional vector $\mathbf{x}$. Each thread was responsible for computing m/t components of the product vector $\mathbf{y} = A\mathbf{x}$. (As usual, t is the number of threads.) The data structures representing A, $\mathbf{x}$, $\mathbf{y}$, m, and n were all shared. For ease of reference, we reproduce the code in Program 4.13.

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

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

Since $S \le t$, $E \le 1$. Table 4.5 shows the run-times and efficiencies of our matrix-vector multiplication with different sets of data and differing numbers of threads.

In each case, the total number of floating point additions and multiplications is 64,000,000; so an analysis that only considers arithmetic operations would predict that a single thread running the code would take the same amount of time for all three inputs. However, it's clear that this is *not* the case. With one thread, the $8{,}000{,}000 \times 8$ system requires about 14% more time than the 8000×8000 system,

```
 1  void *Pth_mat_vect(void* rank) {
 2     long my_rank = (long) rank;
 3     int i, j;
 4     int local_m = m/thread_count;
 5     int my_first_row = my_rank*local_m;
 6     int my_last_row = (my_rank+1)*local_m − 1;
 7
 8     for (i = my_first_row; i <= my_last_row; i++) {
 9        y[i] = 0.0;
10        for (j = 0; j < n; j++)
11            y[i] += A[i][j]*x[j];
12     }
13
14     return NULL;
15  }  /* Pth_mat_vect */
```

Program 4.13: Pthreads matrix-vector multiplication.

Table 4.5 Run-times and efficiencies of matrix-vector multiplication (times are in seconds).

	Matrix Dimension					
	8,000,000 × 8		8000 × 8000		8 × 8,000,000	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.393	1.000	0.345	1.000	0.441	1.000
2	0.217	0.906	0.188	0.918	0.300	0.735
4	0.139	0.707	0.115	0.750	0.388	0.290

and the 8 × 8,000,000 system requires about 28% more time than the 8000 × 8000 system. Both of these differences are at least partially attributable to cache performance

Recall that a *write-miss* occurs when a core tries to update a variable that's not in the cache, and it has to access main memory. A cache profiler (such as Valgrind [51]) shows that when the program is run with the 8,000,000 × 8 input, it has far more cache write-misses than either of the other inputs. The bulk of these occur in Line 9. Since the number of elements in the vector `y` is far greater in this case (8,000,000 vs. 8000 or 8), and each element must be initialized, it's not surprising that this line slows down the execution of the program with the 8,000,000 × 8 input.

Also recall that a *read-miss* occurs when a core tries to read a variable that's not in the cache, and it has to access main memory. A cache profiler shows that when the program is run with the 8 × 8,000,000 input, it has far more cache read-misses than either of the other inputs. These occur in Line 11, and a careful study of this program (see Exercise 4.16) shows that the main source of the differences is due to the reads of x. Once again, this isn't surprising, since for this input, x has 8,000,000 elements, versus only 8000 or 8 for the other inputs.

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

Of more interest to us, though, is the tremendous difference in efficiency as the number of threads is increased. The two-thread efficiency of the program with the $8 \times 8{,}000{,}000$ input is nearly 20% less than the efficiency of the program with the $8{,}000{,}000 \times 8$ and the 8000×8000 inputs. The four-thread efficiency of the program with the $8 \times 8{,}000{,}000$ input is nearly 60% less than the program's efficiency with the $8{,}000{,}000 \times 8$ input and *more* than 60% less than the program's efficiency with the 8000×8000 input. These dramatic decreases in efficiency are even more remarkable when we note that with one thread the program is much slower with $8 \times 8{,}000{,}000$ input. Therefore the numerator in the formula for the efficiency:

$$\text{Parallel Efficiency} = \frac{\text{Serial Run-Time}}{(\text{Number of Threads}) \times (\text{Parallel Run-Time})}$$

will be much larger. Why, then, is the multithreaded performance of the program so much worse with the $8 \times 8{,}000{,}000$ input?

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

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

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

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

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

Why is false sharing not a problem with the other inputs? Let's look at what happens with the 8000 × 8000 input. Suppose thread 2 is assigned to one of the processors and thread 3 is assigned to another. (We don't actually know which threads are assigned to which processors, but it turns out—see Exercise 4.17—that it doesn't matter.) Thread 2 is responsible for computing

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

and thread 3 is responsible for computing

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

If a cache line contains 8 consecutive **double**s, the only possibility for false sharing is on the interface between their assigned elements. If, for example, a single cache line contains

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

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

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

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

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

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

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

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

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

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

This brings up the question of how we might avoid false sharing in our matrix-vector multiplication program. One possible solution is to "pad" the `y` vector with dummy elements to ensure that any update by one thread won't affect another thread's cache line. Another alternative is to have each thread use its own private storage during the multiplication loop, and then update the shared storage when they're done. See Exercise 4.19.

4.11 Thread-safety[9]

Let's look at another potential problem that occurs in shared-memory programming: *thread-safety*. A block of code is **thread-safe** if it can be simultaneously executed by multiple threads without causing problems.

As an example, suppose we want to use multiple threads to "tokenize" a file. Let's suppose that the file consists of ordinary English text, and that the tokens are just contiguous sequences of characters separated from the rest of the text by white space—a space, a tab, or a newline. A simple approach to this problem is to divide the input file into lines of text and assign the lines to the threads in a round-robin fashion: the first line goes to thread 0, the second goes to thread 1, . . . , the tth goes to thread t, the $t+1$st goes to thread 0, and so on.

We can serialize access to the lines of input using semaphores. Then, after a thread has read a single line of input, it can tokenize the line. One way to do this is to use the `strtok` function in `string.h`, which has the following prototype:

```
char* strtok(
        char*        string      /* in/out */,
        const char*  separators  /* in     */);
```

Its usage is a little unusual: the first time it's called the `string` argument should be the text to be tokenized, so in our example it should be the line of input. For subsequent calls, the first argument should be `NULL`. The idea is that in the first call, `strtok` caches a pointer to `string`, and for subsequent calls it returns successive tokens taken from the cached copy. The characters that delimit tokens should be passed in `separators`. We should pass in the string `" \t\n"` as the `separators` argument.

Given these assumptions, we can write the thread function shown in Program 4.14. The main thread has initialized an array of t semaphores—one for each thread. Thread 0's semaphore is initialized to 1. All the other semaphores are initialized to 0. So the code in Lines 9 to 11 will force the threads to sequentially access the lines of input. Thread 0 will immediately read the first line, but all the other threads will block in `sem_wait`. When thread 0 executes the `sem_post`, thread 1 can read a line

[9] This material is also covered in Chapter 5. So if you've already read that chapter, you may want to skim this section.

```
 1  void *Tokenize(void* rank) {
 2     long my_rank = (long) rank;
 3     int count;
 4     int next = (my_rank + 1) % thread_count;
 5     char *fg_rv;
 6     char my_line[MAX];
 7     char *my_string;
 8
 9     sem_wait(&sems[my_rank]);
10     fg_rv = fgets(my_line, MAX, stdin);
11     sem_post(&sems[next]);
12     while (fg_rv != NULL) {
13        printf("Thread %ld > my line = %s", my_rank, my_line);
14
15        count = 0;
16        my_string = strtok(my_line, " \t\n");
17        while ( my_string != NULL ) {
18           count++;
19           printf("Thread %ld > string %d = %s\n",
20                 my_rank, count, my_string);
21           my_string = strtok(NULL, " \t\n");
22        }
23
24        sem_wait(&sems[my_rank]);
25        fg_rv = fgets(my_line, MAX, stdin);
26        sem_post(&sems[next]);
27     }
28
29     return NULL;
30  }  /* Tokenize */
```

Program 4.14: A first attempt at a multithreaded tokenizer.

of input. After each thread has read its first line of input (or end-of-file), any additional input is read in lines 24 to 26. The `fgets` function reads a single line of input and lines 15 to 22 identify the tokens in the line. When we run the program with a single thread, it correctly tokenizes the input stream. The first time we run it with two threads and the input

Pease porridge hot.
Pease porridge cold.
Pease porridge in the pot
Nine days old.

the output is also correct. However, the second time we run it with this input, we get the following output:

```
Thread 0 > my line = Pease porridge hot.
Thread 0 > string 1 = Pease
Thread 0 > string 2 = porridge
Thread 0 > string 3 = hot.
Thread 1 > my line = Pease porridge cold.
Thread 0 > my line = Pease porridge in the pot
Thread 0 > string 1 = Pease
Thread 0 > string 2 = porridge
Thread 0 > string 3 = in
Thread 0 > string 4 = the
Thread 0 > string 5 = pot
Thread 1 > string 1 = Pease
Thread 1 > my line = Nine days old.
Thread 1 > string 1 = Nine
Thread 1 > string 2 = days
Thread 1 > string 3 = old.
```

What happened? Recall that `strtok` caches the input line. It does this by declaring a variable to have **static** storage class. This causes the value stored in this variable to persist from one call to the next. Unfortunately for us, this cached string is shared, not private. Thus thread 0's call to `strtok` with the third line of the input has apparently overwritten the contents of thread 1's call with the second line.

The `strtok` function is *not* thread-safe: if multiple threads call it simultaneously, the output it produces may not be correct. Regrettably, it's not uncommon for C library functions to fail to be thread-safe. For example, neither the random number generator `rand` in `stdlib.h` nor the time conversion function `localtime` in `time.h` is guaranteed to be thread-safe. In some cases, the C standard specifies an alternate, thread-safe, version of a function. In fact, there is a thread-safe version of `strtok`:

```
char* strtok_r(
      char*          string        /* in/out */,
      const char*    separators    /* in     */,
      char**         saveptr_p     /* in/out */);
```

The "`_r`" indicates the function is *reentrant*, which is sometimes used as a synonym for thread-safe.[10] The first two arguments have the same purpose as the arguments to `strtok`. The `saveptr_p` argument is used by `strtok_r` for keeping track of where the

10 However, the distinction is a bit more nuanced; being reentrant means a function can be interrupted and called again (reentered) in different parts of a program's control flow and still execute correctly. This can happen due to nested calls to the function or a trap/interrupt sent from the operating system. Since `strtok` uses a single static pointer to track its state while parsing, multiple calls to the function from different parts of a program's control flow will corrupt the string—therefore it is *not* reentrant. It's worth noting that although reentrant functions, such as `strtok_r`, can also be thread safe, there is no guarantee a reentrant function will *always* be thread safe—and vice versa. It's best to consult the documentation if there's any doubt.

function is in the input string; it serves the purpose of the cached pointer in `strtok`. We can correct our original `Tokenize` function by replacing the calls to `strtok` with calls to `strtok_r`. We simply need to declare a **char**∗ variable to pass in for the third argument, and replace the calls in line 16 and line 21 with the calls

```
my_string = strtok_r(my_line, " \t\n", &saveptr);
. . .
my_string = strtok_r(NULL, " \t\n", &saveptr);
```

respectively.

4.11.1 Incorrect programs can produce correct output

Notice that our original version of the tokenizer program shows an especially insidious form of program error: the first time we ran it with two threads, the program produced correct output. It wasn't until a later run that we saw an error. This, unfortunately, is not a rare occurrence in parallel programs. It's especially common in shared-memory programs. Since, for the most part, the threads are running independently of each other, as we noted earlier, the exact sequence of statements executed is nondeterministic. For example, we can't say when thread 1 will first call `strtok`. If its first call takes place after thread 0 has tokenized its first line, then the tokens identified for the first line should be correct. However, if thread 1 calls `strtok` before thread 0 has finished tokenizing its first line, it's entirely possible that thread 0 may not identify all the tokens in the first line. Therefore it's especially important in developing shared-memory programs to resist the temptation to assume that since a program produces correct output, it must be correct. We always need to be wary of race conditions.

4.12 Summary

Like MPI, Pthreads is a library of functions that programmers can use to implement parallel programs. Unlike MPI, Pthreads is used to implement shared-memory parallelism.

A **thread** in shared-memory programming is analogous to a process in distributed-memory programming. However, a thread is often lighter-weight than a full-fledged process.

We saw that in Pthreads programs, all the threads have access to global variables, while local variables usually are private to the thread running the function. To use Pthreads, we should include the `pthread.h` header file, and, when we compile our program, it may be necessary to link our program with the Pthread library by adding `−lpthread` to the command line. We saw that we can use the functions `pthread_create` and `pthread_join`, respectively, to start and stop a thread function.

When multiple threads are executing, the order in which the statements are executed by the different threads is usually nondeterministic. When nondeterminism results from multiple threads attempting to access a shared resource, such as a shared variable or a shared file, at least one of the accesses is an update, and the accesses can result in an error, we have a **race condition**. One of our most important tasks in writing shared-memory programs is identifying and correcting race conditions. A **critical section** is a block of code that updates a shared resource that can only be updated by one thread at a time, so the execution of code in a critical section should, effectively, be executed as serial code. Thus we should try to design our programs so that they use them as infrequently as possible, and the critical sections we do use should be as short as possible.

We looked at three basic approaches to avoiding conflicting access to critical sections: busy-waiting, mutexes, and semaphores. **Busy-waiting** can be done with a flag variable and a **while** loop with an empty body. It can be very wasteful of CPU cycles. It can also be unreliable if compiler optimization is turned on, so mutexes and semaphores are generally preferable.

A **mutex** can be thought of as a lock on a critical section, since mutexes arrange for *mutually exclusive* access to a critical section. In Pthreads, a thread attempts to obtain a mutex with a call to `pthread_mutex_lock`, and it relinquishes the mutex with a call to `pthread_mutex_unlock`. When a thread attempts to obtain a mutex that is already in use, it *blocks* in the call to `pthread_mutex_lock`. This means that it remains idle in the call to `pthread_mutex_lock` until the system gives it the lock.

A **semaphore** is an **unsigned int** together with two operations: `sem_wait` and `sem_post`. If the semaphore is positive, a call to `sem_wait` simply decrements the semaphore, but if the semaphore is zero, the calling thread blocks until the semaphore is positive, at which point the semaphore is decremented and the thread returns from the call. The `sem_post` operation increments the semaphore; a semaphore can be used as a mutex with `sem_wait` corresponding to `pthread_mutex_lock` and `sem_post` corresponding to `pthread_mutex_unlock`. However, semaphores are more powerful than mutexes, since they can be initialized to any nonnegative value. Furthermore, since there is no "ownership" of a semaphore, any thread can "unlock" a locked semaphore. We saw that semaphores can be easily used to implement **producer–consumer synchronization**. In producer–consumer synchronization, a "consumer" thread waits for some condition or data created by a "producer" thread before proceeding. Semaphores are not part of Pthreads. To use them, we need to include the `semaphore.h` header file.

A **barrier** is a point in a program at which the threads block until all of the threads have reached it. We saw several different means for constructing barriers. One of them used a **condition variable**. A condition variable is a special Pthreads object that can be used to suspend execution of a thread until a condition has occurred. When the condition has occurred, another thread can awaken the suspended thread with a condition signal or a condition broadcast.

The last Pthreads construct we looked at was a **read-write lock**. A read-write lock is used when it's safe for multiple threads to simultaneously *read* a data structure, but if a thread needs to modify or *write* to the data structure, then only that thread can access the data structure during the modification.

We recalled that modern microprocessor architectures use caches to reduce memory access times, so typical architectures have special hardware to ensure that the caches on the different chips are **coherent**. Since the unit of cache coherence, a **cache line** or **cache block**, is usually larger than a single word of memory, this can have the unfortunate side effect that two threads may be accessing different memory locations, but when the two locations belong to the same cache line, the cache-coherence hardware acts as if the threads were accessing the same memory location. Thus, if one of the threads updates its memory location, and then the other thread tries to read its memory location, it will have to retrieve the value from main memory. That is, the hardware is forcing the thread to act as if it were actually sharing the memory location. Hence, this is called **false sharing**, and it can seriously degrade the performance of a shared-memory program.

Some C functions cache data between calls by declaring variables to be **static** . This can cause errors when multiple threads call the function; since static storage is shared among the threads, one thread can overwrite another thread's data. Such a function is not **thread-safe**, and, unfortunately, there are several such functions in the C library. Sometimes, however, there is a thread-safe variant.

When we looked at the program that used the function that wasn't thread-safe, we saw a particularly insidious problem: when we ran the program with multiple threads and a fixed set of input, it sometimes produced correct output, even though the program was erroneous. This means that even if a program produces correct output during testing, there's no guarantee that it is in fact correct—it's up to us to identify possible race conditions.

4.13 Exercises

4.1 When we discussed matrix-vector multiplication, we assumed that both m and n, the number of rows and the number of columns, respectively, were evenly divisible by t, the number of threads. How do the formulas for the assignments change if this is *not* the case?

4.2 If we decide to physically divide a data structure among the threads, that is, if we decide to make various members local to individual threads, we need to consider at least three issues:

a. How are the members of the data structure used by the individual threads?

b. Where and how is the data structure initialized?

c. Where and how is the data structure used after its members are computed?

We briefly looked at the first issue in the matrix-vector multiplication function. We saw that the entire vector `x` was used by all of the threads, so it seemed pretty clear that it should be shared. However, for both the matrix `A` and the product vector `y`, just looking at (a) seemed to suggest that `A` and `y` should have their components distributed among the threads. Let's take a closer look at this.

What would we have to do to divide `A` and `y` among the threads? Dividing `y` wouldn't be difficult—each thread could allocate a block of memory that could be used for storing its assigned components. Presumably, we could do the same for `A`—each thread could allocate a block of memory for storing its assigned rows. Modify the matrix-vector multiplication program so that it distributes both of these data structures. Can you "schedule" the input and output so that the threads can read in `A` and print out `y`? How does distributing `A` and `y` affect the run-time of the matrix-vector multiplication? (Don't include input or output in your run-time.)

4.3 Recall that the compiler is unaware that an ordinary C program is multi-threaded, and as a consequence, it may make optimizations that can interfere with busy-waiting. (Note that compiler optimizations should *not* affect mutexes, condition variables, or semaphores.) An alternative to completely turning off compiler optimizations is to identify some shared variables with the C keyword **volatile**. This tells the compiler that these variables may be updated by multiple threads and, as a consequence, it shouldn't apply optimizations to statements involving them. As an example, recall our busy-wait solution to the race condition when multiple threads attempt to add a private variable into a shared variable:

```
/* x and flag are shared, y is private            */
/* x and flag are initialized to 0 by main thread */

y = Compute(my_rank);
while (flag != my_rank);
x = x + y;
flag++;
```

It's impossible to tell by looking at this code that the order of the **while** statement and the `x = x + y` statement is important; if this code were single-threaded, the order of these two statements wouldn't affect the outcome of the code. But if the compiler determined that it could improve register usage by interchanging the order of these two statements, the resulting code would be erroneous.

If, instead of defining

```
int flag;
int x;
```

we define

```
int volatile flag;
int volatile x;
```

then the compiler will know that both `x` and `flag` can be updated by other threads, so it shouldn't try reordering the statements.

With the `gcc` compiler, the default behavior is no optimization. You can make certain of this by adding the option `−O0` to the command line. Try running the π calculation program that uses busy-waiting (`pth_pi_busy.c`) without optimization. How does the result of the multithreaded calculation compare to the single-threaded calculation? Now try running it with optimization; if you're using `gcc`, replace the `−O0` option with `−O2`. If you found an error, how many threads did you use?

Which variables should be made volatile in the π calculation? Change these variables so that they're volatile and rerun the program with and without optimization. How do the results compare to the single-threaded program?

4.4 The performance of the π calculation program that uses mutexes remains roughly constant once we increase the number of threads beyond the number of available CPUs. What does this suggest about how the threads are scheduled on the available processors?

4.5 Modify the mutex version of the π calculation program so that the critical section is in the **for** loop. How does the performance of this version compare to the performance of the original busy-wait version? How might we explain this?

4.6 Modify the mutex version of the π calculation program so that it uses a semaphore instead of a mutex. How does the performance of this version compare with the mutex version?

4.7 Modify the Pthreads hello, world program to launch an unlimited number of threads—you can effectively ignore the `thread_count` and instead call `pthread_create` in an infinite loop (e.g., `for (thread = 0; ; thread++)`).

Note that in most cases the program will *not* create an unlimited number of threads; you'll observe that the "Hello from thread" messages stop after some time, depending on the configuration of your system. How many threads were created before the messages stopped?

Observe that while nothing is being printed, the program is still running. To determine why no new threads are being created, check the return value of the call to `pthread_create` (hint: use the `perror` function to get a human-readable description of the problem, or look up the error codes). What is the cause of this bug?

Finally, modify the **for** loop containing `pthread_create` to detach each new thread using `pthread_detach`. How many threads are created now?

4.8 Although producer–consumer synchronization is easy to implement with semaphores, it's also possible to implement it with mutexes. The basic idea is to have the producer and the consumer share a mutex. A flag variable that's initialized to `false` by the main thread indicates whether there's anything to consume. With two threads we'd execute something like this:

```
while (1) {
   pthread_mutex_lock(&mutex);
   if (my_rank == consumer) {
      if (message_available) {
            print message;
            pthread_mutex_unlock(&mutex);
            break;
         }
      } else { /* my_rank == producer */
         create message;
         message_available = 1;
         pthread_mutex_unlock(&mutex);
         break;
      }
      pthread_mutex_unlock(&mutex);
   }
```

So if the consumer gets into the loop first, it will see there's no message available and return to the call to `pthread_mutex_lock`. It will continue this process until the producer creates the message. Write a Pthreads program that implements this version of producer–consumer synchronization with two threads. Can you generalize this so that it works with 2k threads—odd-ranked threads are consumers and even-ranked threads are producers? Can you generalize this so that each thread is both a producer and a consumer? For example, suppose that thread q "sends" a message to thread $(q+1)$ mod t and "receives" a message from thread $(q-1+t)$ mod t? Does this use busy-waiting?

4.9 If a program uses more than one mutex, and the mutexes can be acquired in different orders, the program can **deadlock**. That is, threads may block forever waiting to acquire one of the mutexes. As an example, suppose that a program has two shared data structures—for example, two arrays or two linked lists—each of which has an associated mutex. Further suppose that each data structure can be accessed (read or modified) after acquiring the data structure's associated mutex.

a. Suppose the program is run with two threads. Further suppose that the following sequence of events occurs:

Time	Thread 0	Thread 1
0	`pthread_mutex_lock(&mut0)`	`pthread_mutex_lock(&mut1)`
1	`pthread_mutex_lock(&mut1)`	`pthread_mutex_lock(&mut0)`

What happens?

b. Would this be a problem if the program used busy-waiting (with two flag variables) instead of mutexes?

c. Would this be a problem if the program used semaphores instead of mutexes?

4.10 Some implementations of Pthreads define barriers. The function

```
int pthread_barrier_init(
   pthread_barrier_t*              barrier_p  /* out */,
   const pthread_barrierattr_t*    attr_p     /* in  */,
   unsigned                        count      /* in  */);
```

initializes a barrier object, `barrier_p`. As usual, we'll ignore the second argument and just pass in `NULL`. The last argument indicates the number of threads that must reach the barrier before they can continue. The barrier itself is a call to the function

```
int pthread_barrier_wait(
      pthread_barrier_t*  barrier_p  /* in/out */);
```

As with most other Pthreads objects, there is a destroy function

```
int pthread_barrier_destroy(
      pthread_barrier_t*  barrier_p  /* in/out */);
```

Modify one of the barrier programs from the book's website so that it uses a Pthreads barrier. Find a system with a Pthreads implementation that includes barrier and run your program with various numbers of threads. How does its performance compare to the other implementations?

4.11 Modify one of the programs you wrote in the Programming Assignments that follow so that it uses the scheme outlined in Section 4.8 to time itself. To get the time that has elapsed since some point in the past, you can use the macro `GET_TIME` defined in the header file `timer.h` on the book's website. Note that this will give *wall clock* time, not CPU time. Also note that since it's a macro, it can operate directly on its argument. For example, to implement

```
Store current time in my_start;
```

you would use

```
GET_TIME(my_start);
```

not

```
GET_TIME(&my_start);
```

How will you implement the barrier? How will you implement the following pseudocode?

```
elapsed = Maximum of my_elapsed values;
```

4.12 Give an example of a linked list and a sequence of memory accesses to the linked list in which the following pairs of operations can potentially result in problems:

a. Two deletes executed simultaneously
b. An insert and a delete executed simultaneously
c. A member and a delete executed simultaneously
d. Two inserts executed simultaneously
e. An insert and a member executed simultaneously.

4.13 The linked list operations `Insert` and `Delete` consist of two distinct "phases." In the first phase, both operations search the list for either the position of the new node or the position of the node to be deleted. If the outcome of the first phase so indicates, in the second phase a new node is inserted or an existing node is deleted. In fact, it's quite common for linked list programs to split each of these operations into two function calls. For both operations, the first phase involves only read-access to the list; only the second phase modifies the list. Would it be safe to lock the list using a read-lock for the first phase? And then to lock the list using a write-lock for the second phase? Explain your answer.

4.14 Download the various threaded linked list programs from the website. In our examples, we ran a fixed percentage of searches and split the remaining percentage among inserts and deletes.

a. Rerun the experiments with all searches and all inserts.

b. Rerun the experiments with all searches and all deletes.

Is there a difference in the overall run-times? Is insert or delete more expensive?

4.15 Recall that in C a function that takes a two-dimensional array argument must specify the number of columns in the argument list. Thus it is quite common for C programmers to only use one-dimensional arrays, and to write explicit code for converting pairs of subscripts into a single dimension. Modify the Pthreads matrix-vector multiplication so that it uses a one-dimensional array for the matrix and calls a matrix-vector multiplication function. How does this change affect the run-time?

4.16 Download the source file `pth_mat_vect_rand_split.c` from the book's website. Find a program that does cache profiling (for example, Valgrind [51]) and compile the program according to the instructions in the cache profiler documentation. (with Valgrind you will want a symbol table and full optimization; e.g., `gcc −g −O2 . . .`). Now run the program according to the instructions in the cache profiler documentation, using input $k \times (k \cdot 10^6)$, $(k \cdot 10^3) \times (k \cdot 10^3)$, and $(k \cdot 10^6) \times k$. Choose k so large that the number of level 2 cache misses is of the order 10^6 for at least one of the input sets of data.

a. How many level 1 cache write-misses occur with each of the three inputs?

b. How many level 2 cache write-misses occur with each of the three inputs?

c. Where do most of the write-misses occur? For which input data does the program have the most write-misses? Can you explain why?

d. How many level 1 cache read-misses occur with each of the three inputs?

e. How many level 2 cache read-misses occur with each of the three inputs?

f. Where do most of the read-misses occur? For which input data does the program have the most read-misses? Can you explain why?

g. Run the program with each of the three inputs, but without using the cache profiler. With which input is the program the fastest? With which input is the program the slowest? Can your observations about cache misses help explain the differences? How?

4.17 Recall the matrix-vector multiplication example with the 8000×8000 input. Suppose that the program is run with four threads, and thread 0 and thread 2 are assigned to different processors. If a cache line contains 64 bytes or eight **double**s, is it possible for false sharing between threads 0 and 2 to occur for any part of the vector `y`? Why? What about if thread 0 and thread 3 are assigned to different processors—is it possible for false sharing to occur between them for any part of `y`?

4.18 Recall the matrix-vector multiplication example with an $8 \times 8{,}000{,}000$ matrix. Suppose that **double**s use 8 bytes of memory and that a cache line is 64 bytes. Also suppose that our system consists of two dual-core processors.

a. What is the minimum number of cache lines that are needed to store the vector `y`?

b. What is the maximum number of cache lines that are needed to store the vector `y`?

c. If the boundaries of cache lines always coincide with the boundaries of 8-byte **double**s, in how many different ways can the components of `y` be assigned to cache lines?

d. If we only consider which pairs of threads share a processor, in how many different ways can four threads be assigned to the processors in our computer? Here we're assuming that cores on the same processor share cache.

e. Is there an assignment of components to cache lines and threads to processors that will result in no false sharing in our example? In other words, is it possible that the threads assigned to one processor will have their components of `y` in one cache line, and the threads assigned to the other processor will have their components in a different cache line?

f. How many assignments of components to cache lines and threads to processors are there?

g. Of these assignments, how many will result in no false sharing?

4.19 **a.** Modify the matrix-vector multiplication program so that it pads the vector `y` when there's a possibility of false sharing. The padding should be done so that if the threads execute in lock-step, there's no possibility that a single cache line containing an element of `y` will be shared by two or more threads. Suppose, for example, that a cache line stores eight `doubles` and we run the program with four threads. If we allocate storage for at least 48 **double**s in `y`, then, on each pass through the **for** `i` loop, there's no possibility that two threads will simultaneously access the same cache line.

b. Modify the matrix-vector multiplication so that each thread uses private storage for its part of `y` during the **for** `i` loop. When a thread is done computing its part of `y`, it should copy its private storage into the shared variable.

c. How does the performance of these two alternatives compare to the original program? How do they compare to each other?

4.20 Although `strtok_r` is thread-safe, it has the rather unfortunate property that it gratuitously modifies the input string. Write a tokenizer that is thread-safe and doesn't modify the input string.

4.14 Programming assignments

4.1 Write a Pthreads program that implements the histogram program in Chapter 2.

4.2 Suppose we toss darts randomly at a square dartboard, whose bullseye is at the origin, and whose sides are two feet in length. Suppose also that there's a circle inscribed in the square dartboard. The radius of the circle is 1 foot, and its area is π square feet. If the points that are hit by the darts are uniformly distributed (and we always hit the square), then the number of darts that hit inside the circle should approximately satisfy the equation

$$\frac{\text{number in circle}}{\text{total number of tosses}} = \frac{\pi}{4},$$

since the ratio of the area of the circle to the area of the square is $\pi/4$.

We can use this formula to estimate the value of π with a random number generator:

```
number_in_circle = 0;
for (toss = 0; toss < number_of_tosses; toss++) {
   x = random double between −1 and 1;
   y = random double between −1 and 1;
   distance_squared = x*x + y*y;
   if (distance_squared <= 1)
      number_in_circle++;
}
pi_estimate = 4*number_in_circle
   / ((double) number_of_tosses);
```

This is called a "Monte Carlo" method, since it uses randomness (the dart tosses).

Write a Pthreads program that uses a Monte Carlo method to estimate π. The main thread should read in the total number of tosses and print the estimate. You may want to use **long long int**s for the number of hits in the circle and the number of tosses, since both may have to be very large to get a reasonable estimate of π.

4.3 Write a Pthreads program that implements the trapezoidal rule. Use a shared variable for the sum of all the threads' computations. Use busy-waiting, mutexes, and semaphores to enforce mutual exclusion in the critical section. What advantages and disadvantages do you see with each approach?

4.4 Write a Pthreads program that finds the average time required by your system to create and terminate a thread. Does the number of threads affect the average time? If so, how?

4.5 Write a Pthreads program that implements a "task queue." The main thread begins by starting a user-specified number of threads that immediately go to sleep in a condition wait. The main thread generates blocks of tasks to be carried out by the other threads; each time it generates a new block of tasks, it awakens a thread with a condition signal. When a thread finishes executing its block of tasks, it should return to a condition wait. When the main thread completes generating tasks, it sets a global variable indicating that there will be no more tasks, and awakens all the threads with a condition broadcast. For the sake of explicitness, make your tasks linked list operations.

4.6 Write a Pthreads program that uses two condition variables and a mutex to implement a read-write lock. Download the online linked list program that uses Pthreads read-write locks, and modify it to use your read-write locks. Now compare the performance of the program when readers are given preference with the program when writers are given preference. Can you make any generalizations?

CHAPTER 5

Shared-memory programming with OpenMP

Like Pthreads, OpenMP is an API for shared-memory MIMD programming. The "MP" in OpenMP stands for "multiprocessing," a term that is synonymous with shared-memory MIMD computing. Thus OpenMP is designed for systems in which each thread or process can potentially have access to all available memory, and when we're programming with OpenMP, we view our system as a collection of autonomous cores or CPUs, all of which have access to main memory, as in Fig. 5.1.

Although OpenMP and Pthreads are both APIs for shared-memory programming, they have many fundamental differences. Pthreads requires that the programmer explicitly specify the behavior of each thread. OpenMP, on the other hand, sometimes allows the programmer to simply state that a block of code should be executed in parallel, and the precise determination of the tasks and which thread should execute them is left to the compiler and the run-time system. This suggests a further difference between OpenMP and Pthreads; Pthreads (like MPI) is a library of functions that can be linked to a C program, so any Pthreads program can be used with any C compiler, provided the system has a Pthreads library. OpenMP, on the other hand, requires compiler support for some operations, and hence it's entirely possible that

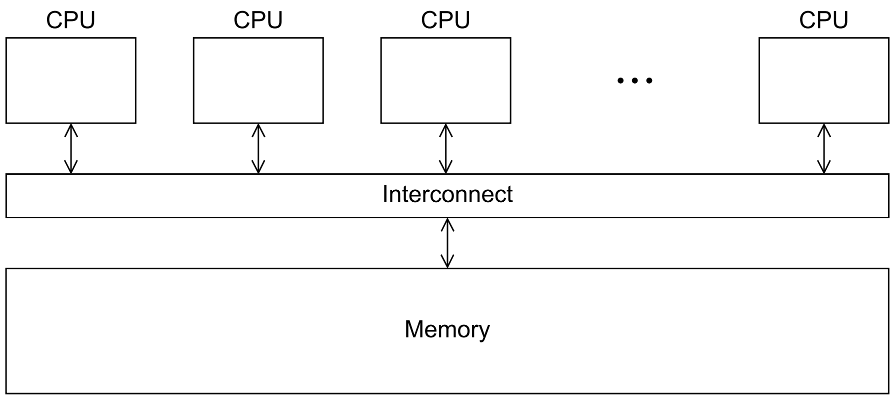

FIGURE 5.1

A shared-memory system.

An Introduction to Parallel Programming. https://doi.org/10.1016/B978-0-12-804605-0.00012-9

you may run across a C compiler that can't compile OpenMP programs into parallel programs.

These differences also suggest why there are two standard APIs for shared-memory programming: Pthreads is lower level and provides us with the power to program virtually any conceivable thread behavior. This power, however, comes with some associated cost—it's up to us to specify every detail of the behavior of each thread. OpenMP, on the other hand, allows the compiler and run-time system to determine some of the details of thread behavior, so it can be simpler to code some parallel behaviors using OpenMP. The cost is that some low-level thread interactions can be more difficult to program.

OpenMP was developed by a group of programmers and computer scientists who believed that writing large-scale high-performance programs using APIs, such as Pthreads, was too difficult, and they defined the OpenMP specification so that shared-memory programs could be developed at a higher level. In fact, OpenMP was explicitly designed to allow programmers to *incrementally* parallelize existing serial programs; this is virtually impossible with MPI and fairly difficult with Pthreads.

In this chapter, we'll learn the basics of OpenMP. We'll learn how to write a program that can use OpenMP, and we'll learn how to compile and run OpenMP programs. Next, we'll learn how to exploit one of the most powerful features of OpenMP: its ability to parallelize serial **for** loops with only small changes to the source code. We'll then look at some other features of OpenMP: task-parallelism and explicit thread synchronization. We'll also look at some standard problems in shared-memory programming: the effect of cache memories on shared-memory programming and problems that can be encountered when serial code—especially a serial library—is used in a shared-memory program.

5.1 Getting started

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 `pragma`s. 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 `pragma`s are free to ignore them. This allows a program that uses the `pragma`s to run on platforms that don't support them. So, in principle, if you have a carefully written OpenMP program, it can be compiled and run on any system with a C compiler, regardless of whether the compiler supports OpenMP. If OpenMP is not supported, then the directives are simply ignored and the code will execute sequentially.

Pragmas in C and C++ start with

```
#pragma
```

As usual, we put the pound sign, #, in column 1, and like other preprocessor directives, we shift the remainder of the directive so that it is aligned with the rest of the code. `Pragma`s (like all preprocessor directives) are, by default, one line in length, so if a pragma won't fit on a single line, the newline needs to be "escaped"—that is, preceded by a backslash \. The details of what follows the **#pragma** depend entirely on which extensions are being used.

Let's take a look at a *very* simple example, a "hello, world" program that uses OpenMP. (See Program 5.1.)

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

void Hello(void);  /* Thread function */

int main(int argc, char* argv[]) {
   /* Get number of threads from command line */
   int thread_count = strtol(argv[1], NULL, 10);

#  pragma omp parallel num_threads(thread_count)
   Hello();

   return 0;
}  /* main */

void Hello(void) {
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   printf("Hello from thread %d of %d\n",
         my_rank, thread_count);

}  /* Hello */
```

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

5.1.1 Compiling and running OpenMP programs

To compile this with gcc we need to include the −fopenmp option[1]:

```
$ 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

[1] Some older versions of gcc may not include OpenMP support. Other compilers will, in general, use different command-line options to specify that the source is an OpenMP program. For details on our assumptions about compiler use, see Section 2.9.

```
$ ./omp_hello 4
```

If we do this, the output might be

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

However, it should be noted that the threads are competing for access to `stdout`, so there's no guarantee that the output will appear in thread-rank order. For example, the output might also be

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

or

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

or any other permutation of the thread ranks.

If we want to run the program with just one thread, we can type

```
$ ./omp_hello 1
```

and we would get the output

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

5.1.2 The program

Let's take a look at the source code. In addition to a collection of directives, OpenMP consists of a library of functions and macros, so we usually need to include a header file with prototypes and macro definitions. The OpenMP header file is `omp.h`, and we include it in Line 3.

In our Pthreads programs, we specified the number of threads on the command line. We'll also usually do this with our OpenMP programs. In Line 9 we therefore use the `strtol` function from `stdlib.h` to get the number of threads. Recall that the syntax of this function is

```
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, a string—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. The return value is the command-line argument converted to a C **long int**.

If you've done a little C programming, there's nothing really new up to this point. When we start the program from the command line, the operating system starts a single-threaded process, and the process executes the code in the `main` function. However, things get interesting in Line 11. This is our first OpenMP directive, and we're using it to specify that the program should start some threads. Each thread should execute the `Hello` function, and when the threads return from the call to `Hello`, they should be terminated, and the process should then terminate when it executes the **return** statement.

That's a lot of bang for the buck (or code). If you studied the Pthreads chapter, you'll recall that we had to write a lot of code to achieve something similar: we needed to allocate storage for a special struct for each thread, we used a **for** loop to start all the threads, and we used another **for** loop to terminate the threads. Thus it's immediately evident that OpenMP provides a higher-level abstraction than Pthreads provides.

We've already seen that `pragma`s in C and C++ start with

```
# pragma
```

OpenMP `pragma`s always begin with

```
# pragma omp
```

Our first directive is a `parallel` directive, and, as you might have guessed, it specifies that the **structured block** of code that follows should be executed by multiple threads. A structured block is a C statement or a compound C statement with one point of entry and one point of exit, although calls to the function `exit` are allowed. This definition simply prohibits code that branches into or out of the middle of the structured block.

Recall that **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—for example, access to `stdin` and `stdout`—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. (See Fig. 5.2.) For more details see Chapters 2 and 4.

At its most basic 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. The algorithm used is fairly complicated; see the OpenMP Standard [47] for details. However, if there are no other threads started, the system will typically run one thread on each available core.

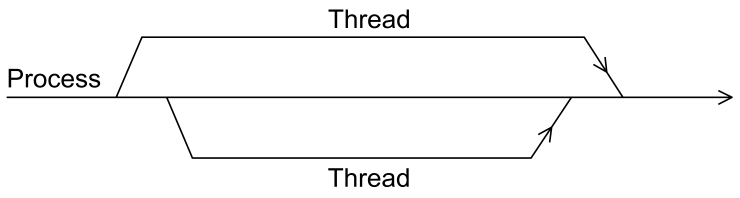

FIGURE 5.2

A process forking and joining two threads.

As we noted earlier, we'll usually specify the number of threads on the command line, so we'll modify our `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. The OpenMP Standard doesn't guarantee that this will actually start `thread_count` threads. However, most current systems can start hundreds or even thousands of threads, so unless we're trying to start *a lot* of threads, we will almost always get the desired number of threads.

What actually happens when the program gets to the `parallel` directive? 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 parlance, the collection of threads executing the `parallel` block—the original thread and the new threads—is called a **team**. OpenMP thread terminology includes the following:

- **master**: the first thread of execution, or *thread 0*.
- **parent**: thread that encountered a `parallel` directive and started a team of threads. In many cases, the parent is also the master thread.
- **child**: each thread started by the parent is considered a *child* thread.

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

When the block of code is completed—in our example, 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 our 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 child threads will terminate and the parent thread will continue executing the code that follows the block. In our example, the parent 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.

As we noted earlier, there is no scheduling of access to `stdout`, so the actual order in which the threads print their results is nondeterministic.

5.1.3 Error checking

To make the code more compact and more readable, our program doesn't do any error checking. Of course, this is dangerous, and, in practice, it's a *very* good idea—one might even say mandatory—to try to anticipate errors and check for them. 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. We might also check that the number of threads actually created by the `parallel` directive is the same as `thread_count`, but in this simple example, this isn't crucial.

A second source of potential problems is the 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 modifications that follow to our program.

Instead of simply including `omp.h`:

```
#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
```

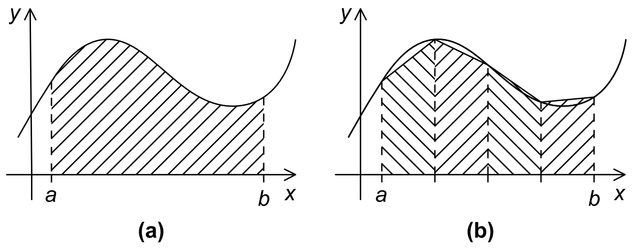

FIGURE 5.3

The trapezoidal rule.

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

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

5.2 The trapezoidal rule

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

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

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

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

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

See Section 3.2.1 for details.

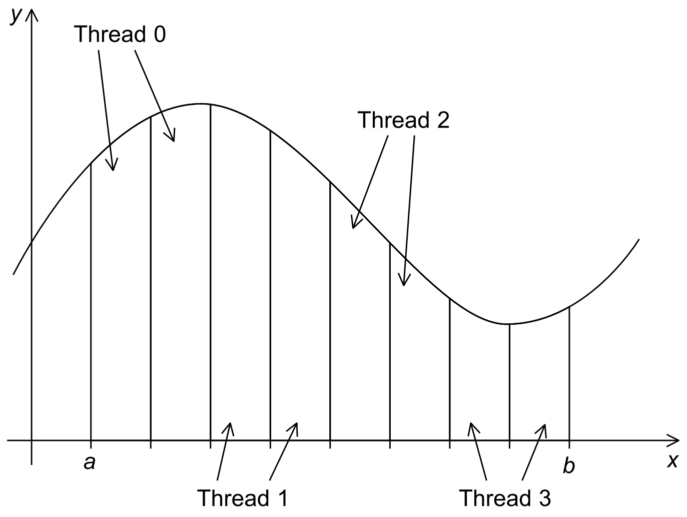

FIGURE 5.4

Assignment of trapezoids to threads.

5.2.1 A first OpenMP version

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

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

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

```
global_result += my_result;
```

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

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

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

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

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

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

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

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

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

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

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

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

int main(int argc, char* argv[]) {
   /* We'll store our result in global_result: */
   double  global_result = 0.0;
   double  a, b;  /* Left and right endpoints   */
   int     n;     /* Total number of trapezoids */
   int     thread_count;

   thread_count = strtol(argv[1], NULL, 10);
   printf("Enter a, b, and n\n");
   scanf("%lf %lf %d", &a, &b, &n);
#  pragma omp parallel num_threads(thread_count)
   Trap(a, b, n, &global_result);

   printf("With n = %d trapezoids, our estimate\n", n);
   printf("of the integral from %f to %f = %.14e\n",
      a, b, global_result);
   return 0;
}  /* main */

void Trap(double a, double b, int n, double* global_result_p) {
   double  h, x, my_result;
   double  local_a, local_b;
   int  i, local_n;
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   h = (b-a)/n;
   local_n = n/thread_count;
   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;
   my_result = (f(local_a) + f(local_b))/2.0;
   for (i = 1; i <= local_n-1; i++) {
     x = local_a + i*h;
     my_result += f(x);
   }
   my_result = my_result*h;

#  pragma omp critical
   *global_result_p += my_result;
}  /* Trap */
```

Program 5.2: First OpenMP trapezoidal rule program.

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

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

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

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

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

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

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

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

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

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

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

So in Line 35, we assign

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

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

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

5.3 Scope of variables

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

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

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

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

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

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

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

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

5.4 The reduction clause

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

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

we would probably define

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

and our function call would be

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

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

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

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

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

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

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

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

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

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

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

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

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

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

is a reduction in which the reduction operator is addition.

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

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

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

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

The syntax of the `reduction` clause is

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

In C, `operator` can be any one of the operators +, $*$, $-$, &, |, ^, &&, $\|$. You may wonder whether the use of subtraction is problematic, though, since subtraction isn't associative or commutative. For example, the serial code

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

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

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

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

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

effectively executes code that is identical to our previous version:

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

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

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

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

5.5 The parallel for directive

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

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

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

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

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

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

Note that it was essential that we made `approx` a reduction variable. If we hadn't, it would have been an ordinary shared variable, and the body of the loop

```
approx += f(a + i*h);
```

would be an unprotected critical section, leading to inconsistent values of `approx`.

However, speaking of scope, the default scope for all variables in a `parallel` directive is shared, but in our `parallel` **for** if the loop variable `i` were shared, the variable update, `i++`, would also be an unprotected critical section. Hence, in a loop that is parallelized with a `parallel` **for** directive the default scope of the loop variable is *private*; in our code, each thread in the team has its own copy of `i`.

5.5.1 Caveats

This is truly wonderful: It may be possible to parallelize a serial program that consists of one large **for** loop by just adding a single `parallel` **for** directive. It may be possible to incrementally parallelize a serial program that has many **for** loops by successively placing `parallel` **for** directives before each loop.

However, things may not be quite as rosy as they seem. There are several caveats associated with the use of the `parallel` **for** directive. First, OpenMP will only parallelize **for** loops—it won't parallelize **while** loops or **do**−**while** loops directly. 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. However, OpenMP will only parallelize **for** loops for which the number of iterations can be determined:

- from the **for** statement itself (that is, the code **for** (. . . ; . . . ; . . .)), and
- prior to execution of the loop.

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. This **for** loop is also not a structured block, since the **break** adds another point of exit from the loop.

In fact, OpenMP will only parallelize **for** loops that are in **canonical form**. Loops in canonical form take one of the forms shown in Program 5.3. The variables and expressions in this template are subject to some fairly obvious restrictions:

```
                                           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    )
```

Program 5.3: Legal forms for parallelizable **for** statements.

- 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.

These restrictions allow the run-time system to determine the number of iterations prior to execution of the loop.

The sole exception to the rule that the run-time system must be able to determine the number of iterations prior to execution is that there *can* be a call to `exit` in the body of the loop.

5.5.2 Data dependences

If a **for** loop fails to satisfy one of the rules outlined in the preceding section, the compiler will simply reject it. For example, suppose we try to compile a program with the following linear search function:

```
1   int Linear_search(int key, int A[], int n) {
2      int i;
3      /* thread_count is global */
4   #  pragma omp parallel for num_threads(thread_count)
5      for (i = 0; i < n; i++)
6         if (A[i] == key) return i;
7      return -1;  /* key not in list */
8   }
```

The `gcc` compiler reports:

```
Line 6: error: invalid exit from OpenMP structured block
```

A more insidious problem occurs in loops in which the computation in one iteration depends on the results of one or more previous iterations. As an example, consider the following code, which computes the first n Fibonacci numbers:

```
fibo[0] = fibo[1] = 1;
for (i = 2; i < n; i++)
   fibo[i] = fibo[i-1] + fibo[i-2];
```

Although we may be suspicious that something isn't quite right, let's try parallelizing the **for** loop with a parallel **for** directive:

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

The compiler will create an executable without complaint. However, if we try running it with more than one thread, we may find that the results are, at best, unpredictable. For example, on one of our systems (if we try using two threads to compute the first 10 Fibonacci numbers), we sometimes get

```
1 1 2 3 5 8 13 21 34 55,
```

which is correct. However, we also occasionally get

```
1 1 2 3 5 8 0 0 0 0.
```

What happened? It appears that the run-time system assigned the computation of fibo[2], fibo[3], fibo[4], and fibo[5] to one thread, while fibo[6], fibo[7], fibo[8], and fibo[9] were assigned to the other. (Remember, the loop starts with i = 2.) In some runs of the program, everything is fine, because the thread that was assigned fibo[2], fibo[3], fibo[4], and fibo[5] finishes its computations before the other thread starts. However, in other runs, the first thread has evidently not computed fibo[4] and fibo[5] when the second computes fibo[6]. It appears that the system has initialized the entries in fibo to 0, and the second thread is using the values fibo[4] = 0 and fibo[5] = 0 to compute fibo[6]. It then goes on to use fibo[5] = 0 and fibo[6] = 0 to compute fibo[7], and so on.

We see two important points here:

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 without using features such as the Tasking API. (See Section 5.10).

The dependence of the computation of fibo[6] on the computation of fibo[5] is called a **data dependence**. Since the value of fibo[5] is calculated in one iteration, and the result is used in a subsequent iteration, the dependence is sometimes called a **loop-carried dependence**.

5.5.3 Finding loop-carried dependences

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

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

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

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

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

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

5.5.4 Estimating π

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

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

We can implement this formula in serial code with

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

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

(Why is it important that `factor` is a **double** instead of an **int** or a **long**?)

How can we parallelize this with OpenMP? We might at first be inclined to do something like this:

```
1         double factor = 1.0;
2         double sum = 0.0;
3  #      pragma omp parallel for num_threads(thread_count) \
4            reduction(+:sum)
5         for (k = 0; k < n; k++) {
6            sum += factor/(2*k+1);
7            factor = -factor;
8         }
9         pi_approx = 4.0*sum;
```

However, it's pretty clear that the update to `factor` in Line 7 in iteration `k` and the subsequent increment of `sum` in Line 6 in iteration `k+1` is an instance of a loop-carried dependence. If iteration `k` is assigned to one thread and iteration `k+1` is assigned to another thread, there's no guarantee that the value of `factor` in Line 6 will be correct. In this case, we can fix the problem by examining the series

$$\sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}.$$

We see that in iteration k, the value of `factor` should be $(-1)^k$, which is $+1$ if k is even and -1 if k is odd, so if we replace the code

```
sum += factor/(2*k+1);
factor = -factor;
```

by

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

or, if you prefer the ?: operator,

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

we will eliminate the loop dependence.

However, things still aren't quite right. If we run the program on one of our systems with just two threads and $n = 1000$, the result is consistently wrong. For example,

```
With n = 1000 terms and 2 threads,
   Our estimate of pi = 2.97063289263385
With n = 1000 terms and 2 threads,
   Our estimate of pi = 3.22392164798593
```

On the other hand, if we run the program with only one thread, we always get

```
With n = 1000 terms and 1 threads,
   Our estimate of pi = 3.14059265383979
```

What's wrong here?

Recall that in a block that has been parallelized by a `parallel` **for** directive, by default any variable declared before the loop—with the sole exception of the loop variable—is shared among the threads. So `factor` is shared and, for example, thread 0 might assign it the value 1, but before it can use this value in the update to `sum`, thread 1 could assign it the value −1. Therefore, in addition to eliminating the loop-carried dependence in the calculation of `factor`, we need to ensure that each thread has its own copy of `factor`. That is, to make our code correct, we need to also ensure that `factor` has private scope. We can do this by adding a `private` clause to the `parallel` **for** directive.

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

The `private` clause specifies that for each variable listed inside the parentheses, a private copy is to be created for each thread. Thus, in our example, each of the `thread_count` threads will have its own copy of the variable `factor`, and hence the updates of one thread to `factor` won't affect the value of `factor` in another thread.

It's important to remember that the value of a variable with private scope is unspecified at the beginning of a `parallel` block or a `parallel` **for** block. Its value is also unspecified after completion of a `parallel` or `parallel` **for** block. So, for example, the output of the first `printf` statement in the following code is unspecified, since it prints the private variable `x` before it's explicitly initialized. Similarly, the output of the final `printf` is unspecified, since it prints `x` after the completion of the `parallel` block.

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

5.5.5 More on scope

Our problem with the variable `factor` is a common one. We usually need to think about the scope of each variable in a `parallel` block or a `parallel` **for** block. Therefore, rather than letting OpenMP decide on the scope of each variable, it's a very good practice for us as programmers to specify the scope of each variable in a block. In fact, OpenMP provides a clause that will explicitly require us to do this: the **default** clause. If we add the clause

```
default(none)
```

to our `parallel` or `parallel` **for** directive, then the compiler will require that we specify the scope of each variable we use in the block and that has been declared outside the block. (Variables that are declared within the block are always private, since they are allocated on the thread's stack.)

For example, using a **default**(`none`) clause, our calculation of π could be written as follows:

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

In this example, we use four variables in the **for** loop. With the default clause, we need to specify the scope of each. As we've already noted, `sum` is a reduction variable (which has properties of both private and shared scope). We've also already noted that `factor` and the loop variable `k` should have private scope. Variables that are never updated in the `parallel` or `parallel` **for** block, such as `n` in this example, can be safely shared. Recall that unlike `private` variables, `shared` variables have the same value in the `parallel` or `parallel` **for** block that they had before the block, and their value after the block is the same as their last value in the block. Thus if `n` were initialized before the block to 1000, it would retain this value in the `parallel` **for** statement, and since the value isn't changed in the **for** loop, it would retain this value after the loop has completed.

5.6 More about loops in OpenMP: sorting

5.6.1 Bubble sort

Recall that the serial *bubble sort* algorithm for sorting a list of integers can be implemented as follows:

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

Here, a stores n **int**s and the algorithm sorts them in increasing order. The outer loop first finds the largest element in the list and stores it in a[n−1]; it then finds the next-to-the-largest element and stores it in a[n−2], and so on. So, effectively, the first pass is working with the full n-element list. The second is working with all of the elements, except the largest; it's working with an $n-1$-element list, and so on.

The inner loop compares consecutive pairs of elements in the current list. When a pair is out of order (a[i] > a[i+1]) it swaps them. This process of swapping will move the largest element to the last slot in the "current" list, that is, the list consisting of the elements

```
a[0], a[1], . . . , a[list_length-1]
```

It's pretty clear that there's a loop-carried dependence in the outer loop; in any iteration of the outer loop the contents of the current list depend on the previous iterations of the outer loop. For example, if at the start of the algorithm a = {3, 4, 1, 2}, then the second iteration of the outer loop should work with the list {3, 1, 2}, since the 4 should be moved to the last position by the first iteration. But if the first two iterations are executing simultaneously, it's possible that the effective list for the second iteration will contain 4.

The loop-carried dependence in the inner loop is also fairly easy to see. In iteration i, the elements that are compared depend on the outcome of iteration $i-1$. If in iteration $i-1$, a[i−1] and a[i] are not swapped, then iteration i should compare a[i] and a[i+1]. If, on the other hand, iteration $i-1$ swaps a[i−1] and a[i], then iteration i should be comparing the original a[i−1] (which is now a[i]) and a[i+1]. For example, suppose the current list is {3,1,2}. Then when $i=1$, we should compare 3 and 2, but if the $i=0$ and the $i=1$ iterations are happening simultaneously, it's entirely possible that the $i=1$ iteration will compare 1 and 2.

It's also not at all clear how we might remove either loop-carried dependence without completely rewriting the algorithm. It's important to keep in mind that even though we can always find loop-carried dependences, it may be difficult or impossible to remove them. The parallel **for** directive is not a universal solution to the problem of parallelizing **for** loops.

Table 5.2 Serial odd-even transposition sort.

Phase	Subscript in Array						
	0		1		2		3
0	9	⇔	7		8	⇔	6
	7		9		6		8
1	7		9	⇔	6		8
	7		6		9		8
2	7	⇔	6		9	⇔	8
	6		7		8		9
3	6		7	⇔	8		9
	6		7		8		9

5.6.2 Odd-even transposition sort

Odd-even transposition sort is a sorting algorithm that's similar to bubble sort, but it has considerably more opportunities for parallelism. Recall from Section 3.7.1 that serial odd-even transposition sort can be implemented as follows:

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

The list `a` stores n **int**s, and the algorithm sorts them into increasing order. During an "even phase" (`phase` % 2 == 0), each odd-subscripted element, `a[i]`, is compared to the element to its "left," `a[i−1]`, and if they're out of order, they're swapped. During an "odd" phase, each odd-subscripted element is compared to the element to its right, and if they're out of order, they're swapped. A theorem guarantees that after n phases, the list will be sorted.

As a brief example, suppose `a = {9, 7, 8, 6}`. Then the phases are shown in Table 5.2. In this case, the final phase wasn't necessary, but the algorithm doesn't bother checking whether the list is already sorted before carrying out each phase.

It's not hard to see that the outer loop has a loop-carried dependence. As an example, suppose as before that `a = {9, 7, 8, 6}`. Then in `phase` 0 the inner loop will compare elements in the pairs $(9, 7)$ and $(8, 6)$, and both pairs are swapped. So for `phase` 1, the list should be `{7, 9, 6, 8}`, and during phase 1 the elements in the pair $(9, 6)$ should be compared and swapped. However, if `phase` 0 and `phase` 1 are executed simultaneously, the pair that's checked in `phase` 1 might be $(7, 8)$, which is in order. Furthermore, it's not clear how one might eliminate this loop-carried dependence, so it would appear that parallelizing the outer **for** loop isn't an option.

The *inner* **for** loops, however, don't appear to have any loop-carried dependences. For example, in an even phase loop variable i will be odd, so for two distinct values of i, say $i = j$ and $i = k$, the pairs $\{j-1, j\}$ and $\{k-1, k\}$ will be disjoint. The comparison and possible swaps of the pairs (`a[j−1]`, `a[j]`) and (`a[k−1]`, `a[k]`) can therefore proceed simultaneously.

```
   for (phase = 0; phase < n; phase++) {
      if (phase % 2 == 0)
#        pragma omp parallel for num_threads(thread_count) \
            default(none) shared(a, n) private(i, tmp)
         for (i = 1; i < n; i += 2) {
            if (a[i−1] > a[i]) {
               tmp = a[i−1];
               a[i−1] = a[i];
               a[i] = tmp;
            }
         }
      else
#        pragma omp parallel for num_threads(thread_count) \
            default(none) shared(a, n) private(i, tmp)
         for (i = 1; i < n−1; i += 2) {
            if (a[i] > a[i+1]) {
               tmp = a[i+1];
               a[i+1] = a[i];
               a[i] = tmp;
            }
         }
   }
```

Program 5.4: First OpenMP implementation of odd-even sort.

Thus we could try to parallelize odd-even transposition sort using the code shown in Program 5.4, but there are a couple of potential problems. First, although any iteration of, say, one even phase doesn't depend on any other iteration of that phase, we've already noted that this is not the case for iterations in `phase` p and `phase` $p+1$. We need to be sure that all the threads have finished `phase` p before any thread starts `phase` $p+1$. However, like the `parallel` directive, the `parallel` **for** directive has an implicit barrier at the end of the loop, so none of the threads will proceed to the next phase, `phase` $p+1$, until all of the threads have completed the current phase, `phase` p.

A second potential problem is the overhead associated with forking and joining the threads. The OpenMP implementation *may* fork and join `thread_count` threads on *each* pass through the body of the outer loop. The first row of Table 5.3 shows run-times for 1, 2, 3, and 4 threads on one of our systems when the input list contained 20,000 elements.

These aren't terrible times, but let's see if we can do better. Each time we execute one of the inner loops, we use the same number of threads, so it would seem to be superior to fork the threads once and reuse the same team of threads for each execution of the inner loops. Not surprisingly, OpenMP provides directives that allow us to do just this. We can fork our team of `thread_count` threads *before* the outer loop with a `parallel` directive. Then, rather than forking a new team of threads with each execution of one of the inner loops, we use a **for** directive, which tells OpenMP to parallelize the **for** loop with the existing team of threads. This modification to the original OpenMP implementation is shown in Program 5.5.

```
#  pragma omp parallel num_threads(thread_count) \
      default(none) shared(a, n) private(i, tmp, phase)
   for (phase = 0; phase < n; phase++) {
      if (phase % 2 == 0)
#        pragma omp for
         for (i = 1; i < n; i += 2) {
            if (a[i-1] > a[i]) {
               tmp = a[i-1];
               a[i-1] = a[i];
               a[i] = tmp;
            }
         }
      else
#        pragma omp for
         for (i = 1; i < n-1; i += 2) {
            if (a[i] > a[i+1]) {
               tmp = a[i+1];
               a[i+1] = a[i];
               a[i] = tmp;
            }
         }
   }
```

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

The **for** directive, unlike the `parallel` **for** directive, doesn't fork any threads. It uses whatever threads have already been forked in the enclosing `parallel` block. There *is* an implicit barrier at the end of the loop. The results of the code—the final list—will therefore be the same as the results obtained from the original parallelized code.

Run-times for this second version of odd-even sort are in the second row of Table 5.3. When we're using two or more threads, the version that uses two **for** directives is at least 17% faster than the version that uses two `parallel` **for** directives, so for this system the slight effort involved in making the change is well worth it.

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

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

5.7 Scheduling loops

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

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

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

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

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

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

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

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

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

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

5.7.1 The `schedule` clause

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

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

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

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

In general, the `schedule` clause has the form

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

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

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

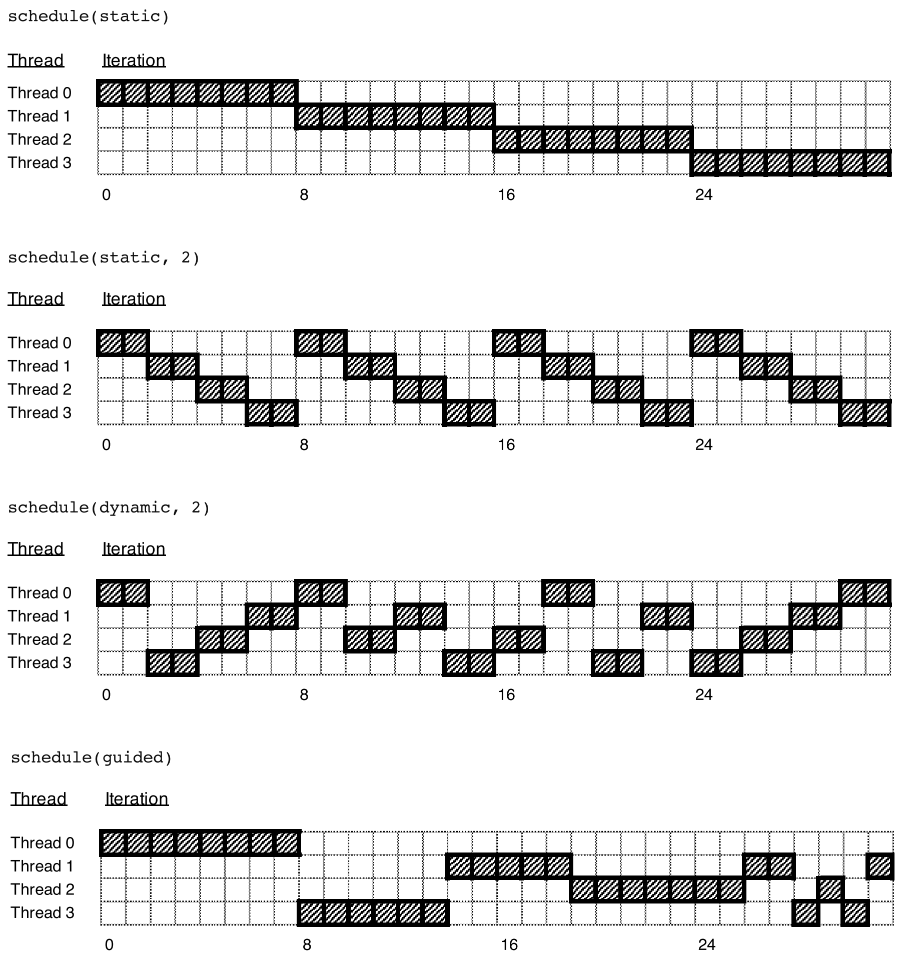

FIGURE 5.5

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

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

5.7.2 The `static` schedule type

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

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

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

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

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

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

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

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

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

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

5.7.3 The `dynamic` and `guided` schedule types

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

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

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

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

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

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

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

5.7.4 The `runtime` schedule type

To understand `schedule(runtime)`, we need to digress for a moment and talk about **environment variables**. As the name suggests, environment variables are named values that can be accessed by a running program. That is, they're available in the program's *environment*. Some commonly used environment variables are PATH, HOME, and SHELL. The PATH variable specifies which directories the shell should search when it's looking for an executable and is usually defined in both Unix and Windows. The HOME variable specifies the location of the user's home directory, and the SHELL variable specifies the location of the executable for the user's shell. These are usually defined in Unix systems. In both Unix-like systems (e.g., Linux and macOS) and Windows, environment variables can be examined and specified on the command line. In Unix-like systems, you can use the shell's command line. In Windows systems, you can use the command line in an integrated development environment.

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

```
$ echo $PATH
```

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

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

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

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

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

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

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

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

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

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

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

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

5.7.5 Which schedule?

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

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

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

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

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

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

5.8 Producers and consumers

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

5.8.1 Queues

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

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

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

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

5.8.2 Message-passing

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

Let's implement a relatively simple message-passing program, in which each thread generates random integer "messages" and random destinations for the messages. After creating the message, the thread enqueues the message in the appropriate message queue. After sending a message, a thread checks its queue to see if it has received a message. If it has, it dequeues the first message in its queue and prints it out. Each thread alternates between sending and trying to receive messages. We'll let the user specify the number of messages each thread should send. When a thread is done sending messages, it receives messages until all the threads are done, at which point all the threads quit. Pseudocode for each thread might look something like this:

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

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

5.8.3 Sending messages

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

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

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

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

5.8.4 Receiving messages

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

Now you may be thinking, "What about the variable storing the size of the queue?" This would be a problem if we simply store the size of the queue. However, if we store two variables, `enqueued` and `dequeued`, then the number of messages in the queue is

```
queue_size = enqueued − dequeued
```

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

Thus we can implement `Try_receive` as follows:

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

5.8.5 Termination detection

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

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

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

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

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

5.8.6 Startup

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

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

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

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

```
#  pragma omp barrier
```

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

5.8.7 The `atomic` directive

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

```
# pragma omp atomic
```

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

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

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

Here `<op>` can be one of the binary operators

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

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

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

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

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

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

5.8.8 Critical sections and locks

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.8.9 Using locks in the message-passing program

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

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

can be replaced with

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

Similarly, the code

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

can be replaced with

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

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

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

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

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

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

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

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

We've already seen some limitations to the use of `critical` directives. However, both named and unnamed `critical` directives are very easy to use. Furthermore, in the implementations of OpenMP that we've used there doesn't seem to be a very large difference between the performance of critical sections protected by a critical directive, and `critical` sections protected by locks, so if you can't use an `atomic` directive, but you can use a `critical` directive, you probably should. Thus the use of locks should probably be reserved for situations in which mutual exclusion is needed for a data structure rather than a block of code.

5.8.11 Some caveats

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Table 5.5 Memory and cache accesses.

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

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

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

```
my_y = x;
```

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

```
x++;
```

Finally, suppose that thread 1 now executes

```
my_z = x;
```

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

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

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

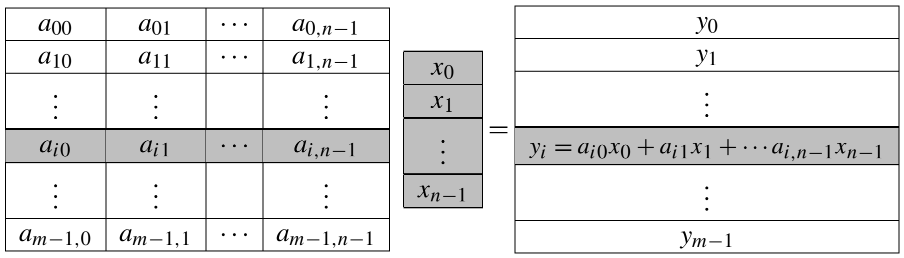
a_{00} a_{01} $\cdots$ $a_{0,n-1}$
a_{10} a_{11} $\cdots$ $a_{1,n-1}$
a_{i0} a_{i1} $\cdots$ $a_{i,n-1}$
$a_{m-1,0}$ $a_{m-1,1}$ $\cdots$ $a_{m-1,n-1}$
x_0 x_1 x_{n-1}
$=$
y_0
y_1
$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}$
y_{m-1}

FIGURE 5.6

Matrix-vector multiplication.

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

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

See Fig. 5.6.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

and thread 3 is responsible for computing

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

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

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

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

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

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

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

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

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

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

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

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

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

5.10 Tasking

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

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

```
#pragma omp task
```

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

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

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

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

```
int fib(int n) {
    int i = 0;
    int j = 0;
    if (n <= 1) {
        // fibs is a global variable
        // It needs storage for n+1 ints
        fibs[n] = n;
        return n;
    }

    i = fib(n − 1);
    j = fib(n − 2);
    fibs[n] = i + j;
    return fibs[n];
}
```

This chain of recursive calls will be time-consuming, so let's execute each as a separate task that can run in parallel. We can do this by adding a `parallel` and a `single` directive before the initial (nonrecursive) call that starts `fib`, and adding **#pragma** omp task before each of the two recursive calls in `fib`. However, after we make this change, the results are incorrect—more specifically, except for `fib`[1], the sequence is all zeroes. This is because the default data scope for variables in tasks is private. So after completing each of the tasks

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

the results in `i` and `j` are lost: `i` and `j` retain their values from the initializations

```
int i = 0;
int j = 0;
```

at the beginning of the function. In other words, the memory locations that are assigned the results of `fib`(`n`−1) and `fib`(`n`−2) are not the same as the memory locations declared at the beginning of the function. So the values that are used to update `fibs`[`n`] are the zeroes assigned at the beginning of the function.

We can adjust the scope of `i` and `j` by declaring the variables to be `shared` in the tasks that execute the recursive call. However executing the program now will produce unpredictable results similar to our original attempt at parallelization. The problem here is that the order in which the various tasks execute isn't specified. In other words, our recursive function calls, `fib`(`n` − 1) and `fib`(`n` − 2) will be run eventually, but the thread executing the task that makes the recursive calls can continue to run and simply **return** the current value of `fibs`[`n`] early. We need to force this task to wait for its subtasks to complete with the `taskwait` directive, which operates as a `barrier` for tasks. We've put this all together in Program 5.6.

```
int fib(int n) {
    int i = 0;
    int j = 0;

    if (n <= 1) {
        fibs[n] = n;
        return n;
    }

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

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

#   pragma omp taskwait
    fibs[n] = i + j;
    return fibs[n];
}
```

Program 5.6: Computing the Fibonacci numbers using OpenMP tasks.

Our parallel Fibonacci program will now produce the correct results, but you may notice significant slowdowns with larger values of n; in fact, there is a good chance that the serial version of the program executes much faster! To gain an intuition as to why this occurs, recall our discussion of the overhead associated with forking and joining threads. Similarly, each task requires its own data environment to be generated upon creation, which takes time. There are a few options we can use to help reduce task creation overhead. The first option is to only create tasks in situations where n is large enough. We can do this with the **if** directive:

```
#pragma omp task shared(i) if(n > 20)
```

which in this case will restrict task creation to only occur when n is larger than 20 (chosen arbitrarily in this case based on some experimentation). Reviewing `fib` again, we can see that there will be a task executing `fib` itself, another executing `fib(n − 1)`, and a third executing `fib(n − 2)` for each recursive call. This is inefficient, because the parent task executing `fib` only launches two subtasks and then simply waits for their results. We can eliminate a task by having the parent thread perform one of the recursive calls to `fib` instead before doing the final calculation after the `taskwait` directive. On our 64-core testbed, these two changes halved the execution time of the program with $n = 35$.

While using the Tasking API requires a bit more planning and care to use—especially with data scoping and limiting runaway task creation—it allows a much broader set of problems to be parallelized by OpenMP.

5.11 Thread-safety[8]

Let's look at another potential problem that occurs in shared-memory programming: *thread-safety*. A block of code is **thread-safe** if it can be simultaneously executed by multiple threads without causing problems.

As an example, suppose we want to use multiple threads to "tokenize" a file. Let's suppose that the file consists of ordinary English text, and that the tokens are just contiguous sequences of characters separated from the rest of the text by white space—a space, a tab, or a newline. A simple approach to this problem is to divide the input file into lines of text and assign the lines to the threads in a round-robin fashion: the first line goes to thread 0, the second goes to thread 1, . . . , the tth goes to thread t, the $t+1$st goes to thread 0, and so on.

We'll read the text into an array of strings, with one line of text per string. Then we can use a `parallel` **for** directive with a `schedule(`**static** ,1) clause to divide the lines among the threads.

One way to tokenize a line is to use the `strtok` function in `string.h`. It has the following prototype:

```
char* strtok(
      char*         string       /* in/out */,
      const char*   separators   /* in     */);
```

Its usage is a little unusual: the first time it's called the `string` argument should be the text to be tokenized, so in our example it should be the line of input. For subsequent calls, the first argument should be `NULL`. The idea is that in the first call, `strtok` caches a pointer to `string`, and for subsequent calls it returns successive tokens taken from the cached copy. The characters that delimit tokens should be passed in `separators`, so we should pass in the string `" \t\n"` as the `separators` argument.

Given these assumptions, we can write the `Tokenize` function shown in Program 5.7. The main function has initialized the array `lines` so that it contains the input text, and `line_count` is the number of strings stored in `lines`. Although for our purposes, we only need the `lines` argument to be an input argument, the `strtok` function modifies its input. Thus when `Tokenize` returns, `lines` will be modified. When we run the program with a single thread, it correctly tokenizes the input stream. The first time we run it with two threads and the input

Pease porridge hot.
Pease porridge cold.
Pease porridge in the pot
Nine days old.

the output is also correct. However, the second time we run it with this input, we get the following output:

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

```
void Tokenize(
      char*  lines[]       /* in/out */,
      int    line_count    /* in     */,
      int    thread_count  /* in     */) {
   int my_rank, i, j;
   char *my_token;

#  pragma omp parallel num_threads(thread_count) \
      default(none) private(my_rank, i, j, my_token) \
      shared(lines, line_count)
   {
      my_rank = omp_get_thread_num();
#     pragma omp for schedule(static, 1)
      for (i = 0; i < line_count; i++) {
         printf("Thread %d > line %d = %s",
               my_rank, i, lines[i]);
         j = 0;
         my_token = strtok(lines[i], " \t\n");
         while ( my_token != NULL ) {
            printf("Thread %d > token %d = %s\n",
                  my_rank, j, my_token);
            my_token = strtok(NULL, " \t\n");
            j++;
         }
      } /* for i */
   }  /* omp parallel */

}  /* Tokenize */
```

Program 5.7: A first attempt at a multi-threaded tokenizer.

```
Thread 0 > line 0 = Pease porridge hot.
Thread 1 > line 1 = Pease porridge cold.
Thread 0 > token 0 = Pease
Thread 1 > token 0 = Pease
Thread 0 > token 1 = porridge
Thread 1 > token 1 = cold.
Thread 0 > line 2 = Pease porridge in the pot
Thread 1 > line 3 = Nine days old.
Thread 0 > token 0 = Pease
Thread 1 > token 0 = Nine
Thread 0 > token 1 = days
Thread 1 > token 1 = old.
```

What happened? Recall that `strtok` caches the input line. It does this by declaring a variable to have `static` storage class. This causes the value stored in this variable to persist from one call to the next. Unfortunately for us, this cached string is shared, not private. Thus it appears that thread 1's call to strtok with the second line has apparently overwritten the contents of thread 0's call with the first line. Even worse, thread 0 has found a token ("days") that should be in thread 1's output.

The strtok function is therefore *not* thread-safe: if multiple threads call it simultaneously, the output it produces may not be correct. Regrettably, it's not uncommon for C library functions to fail to be thread-safe. For example, neither the random number generator rand in stdlib.h nor the time conversion function localtime in time.h is guaranteed to be thread-safe. In some cases, the C standard specifies an alternate, thread-safe version of a function. In fact, there is a thread-safe version of strtok:

```
char* strtok_r(
      char*         string       /* in/out */,
      const char*   separators   /* in     */,
      char**        saveptr_p    /* in/out */);
```

The "_r" is supposed to suggest that the function is *re-entrant*, which is sometimes used as a synonym for thread-safe.[9] The first two arguments have the same purpose as the arguments to strtok. The saveptr argument is used by strtok_r for keeping track of where the function is in the input string; it serves the purpose of the cached pointer in strtok. We can correct our original Tokenize function by replacing the calls to strtok with calls to strtok_r. We simply need to declare a **char**∗ variable to pass in for the third argument, and replace the calls in line 18 and line 22 with the following calls:

```
my_token = strtok_r(lines[i], " \t\n", &saveptr);
. . .
my_token = strtok_r(NULL, " \t\n", &saveptr);
```

respectively.

5.11.1 Incorrect programs can produce correct output

Notice that our original version of the tokenizer program shows an especially insidious form of program error: The first time we ran it with two threads, the program produced correct output. It wasn't until a later run that we saw an error. This, unfortunately, is not a rare occurrence in parallel programs. It's especially common in shared-memory programs. Since, for the most part, the threads are running independently of each other, as we noted back at the beginning of the chapter, the exact

[9] However, the distinction is a bit more nuanced; being reentrant means a function can be interrupted and called again (reentered) in different parts of a program's control flow and still execute correctly. This can happen due to nested calls to the function or a trap/interrupt sent from the operating system. Since strtok uses a single static pointer to track its state while parsing, multiple calls to the function from different parts of a program's control flow will corrupt the string—therefore it is *not* reentrant. It's worth noting that although reentrant functions, such as strtok_r, can also be thread safe, there is no guarantee a reentrant function will *always* be thread safe—and vice versa. It's best to consult the documentation if there's any doubt.

sequence of statements executed is nondeterministic. For example, we can't say when thread 1 will first call `strtok`. If its first call takes place after thread 0 has tokenized its first line, then the tokens identified for the first line should be correct. However, if thread 1 calls `strtok` before thread 0 has finished tokenizing its first line, it's entirely possible that thread 0 may not identify all the tokens in the first line, so it's especially important in developing shared-memory programs to resist the temptation to assume that since a program produces correct output, it must be correct. We always need to be wary of race conditions.

5.12 Summary

OpenMP is a standard for programming shared-memory MIMD systems. It uses both special functions and preprocessor directives called **pragmas**, so unlike Pthreads and MPI, OpenMP requires compiler support. One of the most important features of OpenMP is that it was designed so that developers could *incrementally* parallelize existing serial programs, rather than having to write parallel programs from scratch.

OpenMP programs start multiple **threads** rather than multiple processes. Threads can be much lighter weight than processes; they can share almost all the resources of a process, except each thread must have its own stack and program counter.

To get OpenMP's function prototypes and macros, we include the `omp.h` header in OpenMP programs. There are several OpenMP directives that start multiple threads; the most general is the `parallel` directive:

```
#   pragma omp parallel
    structured block
```

This directive tells the run-time system to execute the following structured block of code in parallel. It may **fork** or start several threads to execute the structured block. A **structured block** is a block of code with a single entry point and a single exit point, although calls to the C library function `exit` are allowed within a structured block. The number of threads started is system dependent, but most systems will start one thread for each available core. The collection of threads executing `block of code` is called a **team**. One of the threads in the team is the thread that was executing the code before the `parallel` directive. This thread is called the **parent**. The additional threads started by the `parallel` directive are called **child** threads. When all of the threads are finished, the child threads are terminated or **joined**, and the parent thread continues executing the code beyond the structured block.

Many OpenMP directives can be modified by **clauses**. We made frequent use of the `num_threads` clause. When we use an OpenMP directive that starts a team of threads, we can modify it with the `num_threads` clause so that the directive will start the number of threads we desire.

When OpenMP starts a team of threads, each of the threads is assigned a rank or ID in the range $0, 1, \ldots,$ `thread_count` $- 1$. The OpenMP library function `omp_get_thread_num` the returns the calling thread's rank. The function `omp_get_num_threads` returns the number of threads in the current team.

A major problem in the development of shared-memory programs is the possibility of **race conditions**. A race condition occurs when multiple threads attempt to access a shared resource, at least one of the accesses is an update, and the accesses can result in an error. Code that is executed by multiple threads that update a shared resource that can only be updated by one thread at a time is called a **critical section**. Thus if multiple threads try to update a shared variable, the program has a race condition, and the code that updates the variable is a critical section. OpenMP provides several mechanisms for ensuring **mutual exclusion** in critical sections. We examined four of them:

1. `Critical` directives ensure that only one thread at a time can execute the structured block. If multiple threads try to execute the code in the critical section, all but one of them will block before the critical section. When one thread finishes the critical section, another thread will be unblocked and enter the code.
2. Named `critical` directives can be used in programs having different critical sections that can be executed concurrently. Multiple threads trying to execute code in critical section(s) with the same name will be handled in the same way as multiple threads trying to execute an unnamed critical section. However, threads entering critical sections with different names can execute concurrently.
3. An `atomic` directive can only be used when the critical section has the form `x <op>=<expression>`, `x++`, `++x`, `x−−`, or `−−x`. It's designed to exploit special hardware instructions, so it can be much faster than an ordinary critical section.
4. Simple locks are the most general form of mutual exclusion. They use function calls to restrict access to a critical section:

   ```
   omp_set_lock(&lock);
   critical section
   omp_unset_lock(&lock);
   ```

   When multiple threads call `omp_set_lock`, only one of them will proceed to the critical section. The others will block until the first thread calls `omp_unset_lock`. Then one of the blocked threads can proceed.

All of the mutual exclusion mechanisms can cause serious program problems, such as deadlock, so they need to be used with great care.

A **for** directive can be used to partition the iterations in a **for** loop among the threads. This directive doesn't start a team of threads; it divides the iterations in a **for** loop among the threads in an existing team. If we want also to start a team of threads, we can use the `parallel` **for** directive. There are a number of restrictions on the form of a **for** loop that can be parallelized; basically, the run-time system must be able to determine the total number of iterations through the loop body before the loop begins execution. For details, see Program 5.3.

It's not enough, however, to ensure that our **for** loop has one of the canonical forms. It must also not have any **loop-carried dependences**. A loop-carried dependence occurs when a memory location is read or written in one iteration and written in another iteration. OpenMP won't detect loop-carried dependences; it's up to us, the programmers, to detect and eliminate them. It may, however, be impossible to eliminate them, in which case the loop isn't a candidate for parallelization.

By default, most systems use a **block partitioning** of the iterations in a parallelized **for** loop. If there are *n* iterations, this means that roughly the first $n/thread\_count$ are assigned to thread 0, the next $n/thread\_count$ are assigned to thread 1, and so on. However, there are a variety of **scheduling** options provided by `OpenMP`. The `schedule` clause has the form

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

The `type` can be `static`, `dynamic`, `guided`, `auto`, or `runtime`. In a `static` schedule, the iterations can be assigned to the threads before the loop starts execution. In `dynamic` and `guided` schedules, the iterations are assigned on the fly. When a thread finishes a chunk of iterations—a contiguous block of iterations—it requests another chunk. If **auto** is specified, the schedule is determined by the compiler or run-time system, and if `runtime` is specified, the schedule is determined at run-time by examining the environment variable `OMP_SCHEDULE`.

Only `static`, `dynamic`, and `guided` schedules can have a `chunksize`. In a `static` schedule, the chunks of `chunksize` iterations are assigned in round robin fashion to the threads. In a `dynamic` schedule, each thread is assigned `chunksize` iterations, and when a thread completes its chunk, it requests another chunk. In a `guided` schedule, the size of the chunks decreases as the iteration proceeds.

In OpenMP the **scope** of a variable is the collection of threads to which the variable is accessible. Typically, any variable that was defined before the OpenMP directive has **shared** scope within the construct. That is, all the threads have access to it. There are a couple of important exceptions to this. The first is that the loop variable in a **for** or `parallel` **for** construct is **private**, that is, each thread has its own copy of the variable. The second exception is that variables used in a `task` construct have private scope. Variables that are defined within an OpenMP construct have private scope, since they will be allocated from the executing thread's stack.

As a rule of thumb, it's a good idea to explicitly assign the scope of variables. This can be done by modifying a `parallel` or `parallel` **for** directive with the *scoping* clause:

```
        default(none)
```

This tells the system that the scope of every variable that's used in the OpenMP construct must be explicitly specified. Most of the time this can be done with `private` or `shared` clauses.

The only exceptions we encountered were **reduction variables**. A **reduction operator** is an associative binary operation (such as addition or multiplication), and a **reduction** is a computation that repeatedly applies the same reduction operator to a sequence of operands to get a single result. Furthermore, all of the intermediate results of the operation should be stored in the same variable: the **reduction variable**. For example, if *A* is an array with *n* elements, then the code

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

is a reduction. The reduction operator is addition and the reduction variable is `sum`. If we try to parallelize this loop, the reduction variable should have properties of both private and shared variables. Initially we would like each thread to add its array elements into its own private `sum`, but when the threads are done, we want the private `sum`'s combined into a single, shared `sum`. OpenMP therefore provides the `reduction` clause for identifying reduction variables and operators.

A `barrier` directive will cause the threads in a team to block until all the threads have reached the directive. We've seen that the `parallel`, `parallel` **for**, and **for** directives have implicit barriers at the end of the structured block.

We recalled that modern microprocessor architectures use caches to reduce memory access times, so typical architectures have special hardware to ensure that the caches on the different chips are **coherent**. Since the unit of cache coherence, a **cache line** or **cache block**, is usually larger than a single word of memory, this can have the unfortunate side effect that two threads may be accessing different memory locations, but when the two locations belong to the same cache line, the cache-coherence hardware acts as if the threads were accessing the same memory location—if one of the threads updates its memory location, and then the other thread tries to read its memory location, it will have to retrieve the value from main memory. That is, the hardware is forcing the thread to act as if it were actually sharing the memory location. Hence, this is called **false sharing**, and it can seriously degrade the performance of a shared-memory program.

The `task` directive can be used to get parallel threads to execute different sequences of instructions. For example, if our code contains three functions, `f1`, `f2`, and `f3` that are independent of each other, then the serial code

```
   f1(args1);
   f2(args2);
   f3(args3);
```

can be parallelized by using the following directives:

```
#  pragma omp parallel numthreads(3)
#     pragma omp single
      {
#        pragma omp task
         f1(args1);

#        pragma omp task
         f2(args2);

#        pragma omp task
         f3(args3);
      }
```

The `parallel` directive will start three threads. If we omitted the `single` directive, each function call would be executed by all three threads. The `single` directive ensures that only one of the threads calls each of the functions. If each of the functions returns a value and we want to (say) add these values, we can put a barrier before the addition by adding a `taskwait` directive:

```
#  pragma omp parallel numthreads(3)
#     pragma omp single
      {
#        pragma omp task shared(x1)
         x1 = f1(args1);

#        pragma omp task shared(x2)
         x2 = f2(args2);

#        pragma omp task shared(x3)
         x3 = f3(args3);

#        pragma omp taskwait
         x = x1 + x2 + x3;
      }
```

Some C functions cache data between calls by declaring variables to be `static`. This can cause errors when multiple threads call the function; since static storage is shared among the threads, one thread can overwrite another thread's data. Such a function is not **thread-safe**, and, unfortunately, there are several such functions in the C library. Sometimes, however, the library has a thread-safe variant of a function that isn't thread-safe.

In one of our programs, we saw a particularly insidious problem: when we ran the program with multiple threads and a fixed set of input, it sometimes produced correct output, even though it had an error. Producing correct output during testing doesn't guarantee that the program is in fact correct. It's up to us to identify possible race conditions.

5.13 Exercises

5.1 If it's defined, the `_OPENMP` macro is a decimal **int**. Write a program that prints its value. What is the significance of the value?

5.2 Download `omp_trap_1.c` from the book's website, and delete the `critical` directive. Now compile and run the program with more and more threads and larger and larger values of n. How many threads and how many trapezoids does it take before the result is incorrect?

5.3 Modify `omp_trap1.c` so that

(i) it uses the first block of code on page 234, and

(ii) the time used by the parallel block is timed using the OpenMP function `omp_get_wtime()`. The syntax is

```
double omp_get_wtime(void)
```

It returns the number of seconds that have elapsed since some time in the past. For details on taking timings, see Section 2.6.4. Also recall that OpenMP has a `barrier` directive:

```
#  pragma omp barrier
```

Now on a system with at least two cores, time the program with

a. one thread and a large value of n, and

b. two threads and the same value of n.

What happens? Download `omp_trap_2.c` from the book's website. How does its performance compare? Explain your answers.

5.4 Recall that OpenMP creates private variables for reduction variables, and these private variables are initialized to the identity value for the reduction operator. For example, if the operator is addition, the private variables are initialized to 0, while if the operator is multiplication, the private variables are initialized to 1. What are the identity values for the operators `&&`, `||`, `&`, `|`, `^`?

5.5 Suppose that on the amazing Bleeblon computer, variables with type **float** can store three decimal digits. Also suppose that the Bleeblon's floating point registers can store four decimal digits, and that after any floating point operation, the result is rounded to three decimal digits before being stored. Now suppose a C program declares an array `a` as follows:

```
float a[] = {4.0, 3.0, 3.0, 1000.0};
```

a. What is the output of the following block of code if it's run on the Bleeblon?

```
int i;
float sum = 0.0;
for (i = 0; i < 4; i++)
   sum += a[i];
printf("sum = %4.1f\n", sum);
```

b. Now consider the following code:

```
   int i;
   float sum = 0.0;
#  pragma omp parallel for num_threads(2) \
      reduction(+:sum)
   for (i = 0; i < 4; i++)
      sum += a[i];
   printf("sum = %4.1f\n", sum);
```

Suppose that the run-time system assigns iterations `i` = 0, 1 to thread 0 and `i` = 2, 3 to thread 1. What is the output of this code on the Bleeblon?

5.6 Write an OpenMP program that determines the default scheduling of parallel **for** loops. Its input should be the number of iterations, and its output should be which iterations of a parallelized **for** loop are executed by which thread. For example, if there are two threads and four iterations, the output might be the following:

```
Thread 0: Iterations 0 — 1
Thread 1: Iterations 2 — 3
```

5.7 In our first attempt to parallelize the program for estimating π, our program was incorrect. In fact, we used the result of the program when it was run with one thread as evidence that the program run with two threads was incorrect. Explain why we could "trust" the result of the program when it was run with one thread.

5.8 Consider the loop

```
a[0] = 0;
for (i = 1; i < n; i++)
   a[i] = a[i-1] + i;
```

There's clearly a loop-carried dependence, as the value of a[i] can't be computed without the value of a[i-1]. Can you see a way to eliminate this dependence and parallelize the loop?

5.9 Modify the trapezoidal rule program that uses a parallel **for** directive (omp_trap_3.c) so that the parallel **for** is modified by a schedule(runtime) clause. Run the program with various assignments to the environment variable OMP_SCHEDULE and determine which iterations are assigned to which thread. This can be done by allocating an array iterations of n **int**s and in the Trap function assigning omp_get_thread_num() to iterations[i] in the ith iteration of the **for** loop. What is the default assignment of iterations on your system? How are guided schedules determined?

5.10 Recall that all structured blocks modified by an unnamed critical directive form a single critical section. What happens if we have a number of atomic directives in which different variables are being modified? Are they all treated as a single critical section?

We can write a small program that tries to determine this. The idea is to have all the threads simultaneously execute something like the following code:

```
   int i;
   double my_sum = 0.0;
   for (i = 0; i < n; i++)
#     pragma omp atomic
      my_sum += sin(i);
```

We can do this by modifying the code by a parallel directive:

```
#  pragma omp parallel num_threads(thread_count)
   {
      int i;
      double my_sum = 0.0;
      for (i = 0; i < n; i++)
#           pragma omp atomic
            my_sum += sin(i);
    }
```

Note that since my_sum and i are declared in the parallel block, each thread has its own private copy. Now if we time this code for large n when thread_count $= 1$ and we also time it when thread_count > 1, then as long as thread_count is less than the number of available cores, the run-time for the single-threaded run should be roughly the same as the time for the multithreaded run if the different threads' executions of my_sum += sin(i) are treated as different critical sections. On the other hand, if the different executions of my_sum += sin(i) are all treated as a single critical section, the multithreaded run should be much slower than the single-threaded run. Write an OpenMP program that implements this test. Does your implementation of OpenMP allow simultaneous execution of updates to different variables when the updates are protected by atomic directives?

5.11 Recall that in C, a function that takes a two-dimensional array argument must specify the number of columns in the argument list, so it is quite common for C programmers to only use one-dimensional arrays, and to write explicit code for converting pairs of subscripts into a single dimension. (See Exercise 3.14.) Modify the OpenMP matrix-vector multiplication so that it uses a one-dimensional array for the matrix.

5.12 Download the source file omp_mat_vect_rand_split.c from the book's website. Find a program that does cache profiling (e.g., Valgrind [51]) and compile the program according to the instructions in the cache profiler documentation. (For example, with Valgrind you will want a symbol table and full optimization. With gcc use gcc −g −O2 . . . .) Now run the program according to the instructions in the cache profiler documentation, using input $k \times (k \cdot 10^6)$, $(k \cdot 10^3) \times (k \cdot 10^3)$, and $(k \cdot 10^6) \times k$. Choose k so large that the number of level 2 cache misses is of the order 10^6 for at least one of the input sets of data.

a. How many level 1 cache write-misses occur with each of the three inputs?
b. How many level 2 cache write-misses occur with each of the three inputs?
c. Where do most of the write-misses occur? For which input data does the program have the most write-misses? Can you explain why?
d. How many level 1 cache read-misses occur with each of the three inputs?
e. How many level 2 cache read-misses occur with each of the three inputs?
f. Where do most of the read-misses occur? For which input data does the program have the most read-misses? Can you explain why?
g. Run the program with each of the three inputs but without using the cache profiler. With which input is the program the fastest? With which input is the program the slowest? Can your observations about cache misses help explain the differences? How?

5.13 Recall the matrix-vector multiplication example with the 8000×8000 input. Suppose that thread 0 and thread 2 are assigned to different processors. If a cache line contains 64 bytes or 8 **double**s, is it possible for false sharing between threads 0 and 2 to occur for any part of the vector `y`? Why? What about if thread 0 and thread 3 are assigned to different processors; is it possible for false sharing to occur between them for any part of `y`?

5.14 Recall the matrix-vector multiplication example with an 8 × 8,000,000 matrix. Suppose that **double**s use 8 bytes of memory and that a cache line is 64 bytes. Also suppose that our system consists of two dual-core processors.

a. What is the minimum number of cache lines that are needed to store the vector `y`?

b. What is the maximum number of cache lines that are needed to store the vector `y`?

c. If the boundaries of cache lines always coincide with the boundaries of 8-byte **double**s, in how many different ways can the components of `y` be assigned to cache lines?

d. If we only consider which pairs of threads share a processor, in how many different ways can four threads be assigned to the processors in our computer? Here we're assuming that cores on the same processor share cache.

e. Is there an assignment of components to cache lines and threads to processors that will result in no false-sharing in our example? In other words, is it possible that the threads assigned to one processor will have their components of `y` in one cache line and the threads assigned to the other processor will have their components in a different cache line?

f. How many assignments of components to cache lines and threads to processors are there?

g. Of these assignments, how many will result in no false sharing?

5.15 **a.** Modify the matrix-vector multiplication program so that it pads the vector `y` when there's a possibility of false sharing. The padding should be done so that if the threads execute in lock-step, there's no possibility that a single cache line containing an element of `y` will be shared by two or more threads. Suppose, for example, that a cache line stores eight **double**s and we run the program with four threads. If we allocate storage for at least 48 **double**s in `y`, then, on each pass through the **for** `i` loop, there's no possibility that two threads will simultaneously access the same cache line.

b. Modify the matrix-vector multiplication program so that each thread uses private storage for its part of `y` during the **for** `i` loop. When a thread is done computing its part of `y`, it should copy its private storage into the shared variable.

c. How does the performance of these two alternatives compare to the original program? How do they compare to each other?

5.16 The following code can be used as the basis of an implementation of merge sort:

```
/* Sort elements of list from list[lo] to list[hi] */
void Mergesort(int list[], int lo, int hi) {
   if (lo < hi) {
      int mid = (lo + hi)/2;
      Mergesort(list, lo, mid);
      Mergesort(list, mid+1, hi);
      Merge(list, lo, mid, hi);
   }
}  /* Mergesort */
```

The `Mergesort` function can be called from the following `main` function:

```
int main(int argc, char* argv[])
   int *list, n;

   Get_args(argc, argv, &n);
   list = malloc(n*sizeof(int));
   Get_list(list, n);
   Mergesort(list, 0, n-1);
   Print_list(list, n);
   free(list);
   return 0;
}  /* main */
```

Use the `task` directive (and any other OpenMP directives) to implement a parallel merge sort program.

5.17 Although `strtok_r` is thread-safe, it has the rather unfortunate property that it gratuitously modifies the input string. Write a tokenizer that is thread-safe and doesn't modify the input string.

5.14 Programming assignments

5.1 Use OpenMP to implement the parallel histogram program discussed in Chapter 2.

5.2 Suppose we toss darts randomly at a square dartboard whose bullseye is at the origin and whose sides are two feet in length. Suppose also that there's a circle inscribed in the square dartboard. The radius of the circle is 1 foot, and its area is π square feet. If the points that are hit by the darts are uniformly distributed (and we always hit the square), then the number of darts that hit inside the circle should approximately satisfy the equation

$$\frac{\text{number in circle}}{\text{total number of tosses}} = \frac{\pi}{4},$$

since the ratio of the area of the circle to the area of the square is $\pi/4$.

We can use this formula to estimate the value of π with a random number generator:

```
number_in_circle = 0;
for (toss = 0; toss < number_of_tosses; toss++) {
   x = random double between −1 and 1;
   y = random double between −1 and 1;
   distance_squared = x*x + y*y;
   if (distance_squared <= 1) number_in_circle++;
}

pi_estimate
    = 4*number_in_circle / ((double) number_of_tosses);
```

This is called a "Monte Carlo" method, since it uses randomness (the dart tosses). Write an OpenMP program that uses a Monte Carlo method to estimate π. Read in the total number of tosses before forking any threads. Use a `reduction` clause to find the total number of darts hitting inside the circle. Print the result after joining all the threads. You may want to use **long long int**s for the number of hits in the circle and the number of tosses, since both may have to be very large to get a reasonable estimate of π.

5.3 Count sort is a simple serial sorting algorithm that can be implemented as follows:

```
void Count_sort(int a[], int n) {
   int i, j, count;
   int* temp = malloc(n*sizeof(int));

   for (i = 0; i < n; i++) {
      count = 0;
      for (j = 0; j < n; j++)
         if (a[j] < a[i])
            count++;
         else if (a[j] == a[i] && j < i)
            count++;
      temp[count] = a[i];
   }

   memcpy(a, temp, n*sizeof(int));
   free(temp);
}  /* Count_sort */
```

The basic idea is that for each element `a[i]` in the list `a`, we count the number of elements in the list that are less than `a[i]`. Then we insert `a[i]` into a temporary list using the subscript determined by the count. There's a slight problem with this approach when the list contains equal elements, since they could get assigned to the same slot in the temporary list. The code deals with this by incrementing the count for equal elements on the basis of the subscripts. If both `a[i] == a[j]` *and* `j < i`, then we count `a[j]` as being "less than" `a[i]`.

After the algorithm has completed, we overwrite the original array with the temporary array using the string library function `memcpy`.

a. If we try to parallelize the **for** `i` loop (the outer loop), which variables should be private and which should be shared?

b. If we parallelize the **for** `i` loop using the scoping you specified in the previous part, are there any loop-carried dependences? Explain your answer.

c. Can we parallelize the call to `memcpy`? Can we modify the code so that this part of the function will be parallelizable?

d. Write a C program that includes a parallel implementation of `Count_sort`.

e. How does the performance of your parallelization of `Count_sort` compare to serial `Count_sort`? How does it compare to the serial `qsort` library function?

5.4 Recall that when we solve a large linear system, we often use Gaussian elimination followed by *backward substitution*. Gaussian elimination converts an $n \times n$ linear system into an *upper triangular* linear system by using the "row operations":

- Add a multiple of one row to another row
- Swap two rows
- Multiply one row by a nonzero constant

An upper triangular system has zeroes below the "diagonal" extending from the upper left-hand corner to the lower right-hand corner.
For example, the linear system

$$\begin{array}{rcrcrcl} 2x_0 & - & 3x_1 & & & = & 3 \\ 4x_0 & - & 5x_1 & + & x_2 & = & 7 \\ 2x_0 & - & x_1 & - & 3x_2 & = & 5 \end{array}$$

can be reduced to the upper triangular form

$$\begin{array}{rcrcrcl} 2x_0 & - & 3x_1 & & & = & 3 \\ & & x_1 & + & x_2 & = & 1 \\ & & & - & 5x_2 & = & 0 \end{array},$$

and this system can be easily solved by first finding x_2 using the last equation, then finding x_1 using the second equation, and finally finding x_0 using the first equation.
We can devise a couple of serial algorithms for back substitution. The "row-oriented" version is

```
for (row = n-1; row >= 0; row--) {
   x[row] = b[row];
   for (col = row+1; col < n; col++)
      x[row] -= A[row][col]*x[col];
   x[row] /= A[row][row];
}
```

Here the “right-hand side” of the system is stored in the array b, the two-dimensional array of coefficients is stored in the array A, and the solutions are stored in the array x. An alternative is the following “column-oriented” algorithm:

```
for (row = 0; row < n; row++)
   x[row] = b[row];

for (col = n-1; col >= 0; col--) {
   x[col] /= A[col][col];
   for (row = 0; row < col; row++)
      x[row] -= A[row][col]*x[col];
}
```

- **a.** Determine whether the outer loop of the row-oriented algorithm can be parallelized.
- **b.** Determine whether the inner loop of the row-oriented algorithm can be parallelized.
- **c.** Determine whether the (second) outer loop of the column-oriented algorithm can be parallelized.
- **d.** Determine whether the inner loop of the column-oriented algorithm can be parallelized.
- **e.** Write one OpenMP program for each of the loops that you determined could be parallelized. You may find the single directive useful—when a block of code is being executed in parallel and a sub-block should be executed by only one thread, the sub-block can be modified by a **#pragma** omp single directive. The threads in the executing team will block at the end of the directive until all of the threads have completed it.
- **f.** Modify your parallel loop with a schedule(runtime) clause and test the program with various schedules. If your upper triangular system has 10,000 variables, which schedule gives the best performance?

5.5 Use OpenMP to implement a program that does Gaussian elimination. (See the preceding problem.) You can assume that the input system doesn’t need any row-swapping.

5.6 Use OpenMP to implement a producer consumer program in which some of the threads are producers and others are consumers. The producers read text from a collection of files, one per producer. They insert lines of text into a single shared queue. The consumers take the lines of text and tokenize them. Tokens are “words” separated by white space. When a consumer finds a token, it writes it to stdout.

5.7 Use your answer to Exercise 5.16 to implement a parallel merge sort program. Get the number of threads from the command line. Either get the input list from stdin or use the C library random function to generate the list. How does the performance of the parallel program compare to a serial merge sort?