MODULE 4

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

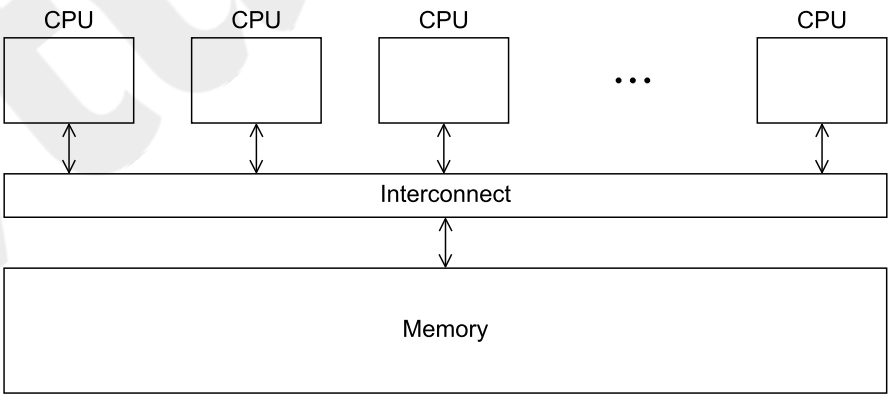

FIGURE 5.1

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.)

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

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

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.

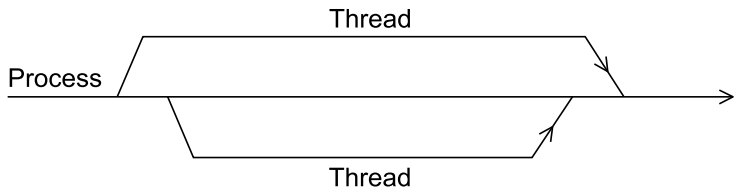

FIGURE 5.2

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 $0, 1, \ldots,$ `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
```

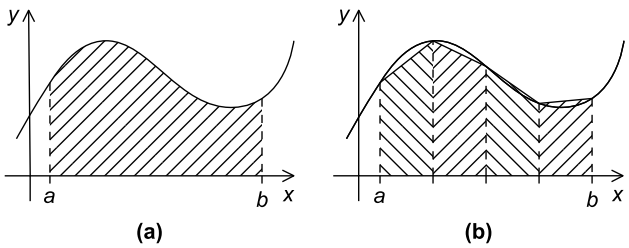

FIGURE 5.3

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.

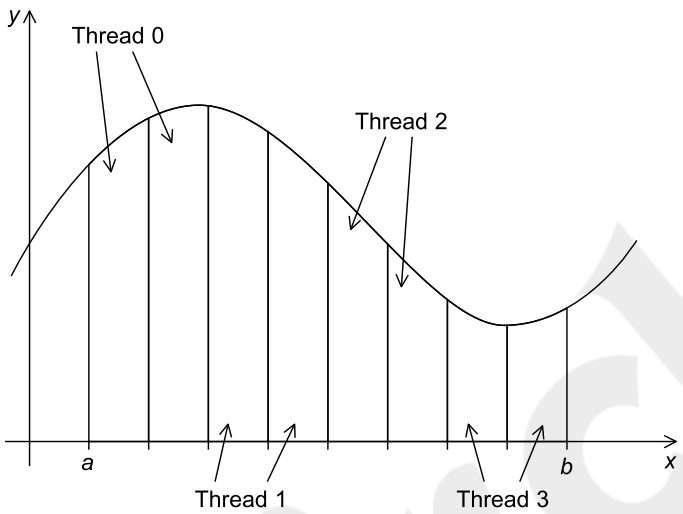

FIGURE 5.4

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.

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

Program 5.2: First OpenMP trapezoidal rule program.

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

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

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

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

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

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

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

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

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

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

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

So in Line 35, we assign

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

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

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

5.3 Scope of variables

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

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

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

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

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

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

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

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

5.4 The reduction clause

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

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

we would probably define

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

and our function call would be

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

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

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

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

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

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

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

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

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

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

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

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

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

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

is a reduction in which the reduction operator is addition.

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

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

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

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

The syntax of the `reduction` clause is

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

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

associative or commutative. For example, the serial code

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

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

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

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

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

effectively executes code that is identical to our previous version:

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

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

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

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

5.5 The parallel for directive

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

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

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

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

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

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

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

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

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

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

5.5.4 Estimating π

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

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

We can implement this formula in serial code with

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

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

(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

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

by

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

or, if you prefer the ?: operator,

```
1            factor = (k % 2 == 0) ? 1.0 : -1.0;
2            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,

```
1     With n = 1000 terms and 2 threads,
2        Our estimate of pi = 2.97063289263385
3     With n = 1000 terms and 2 threads,
4        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;
```

```
9         printf("Thread %d > after initialization, x = %d\n",
10              my_rank, x);
11    }
12    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.

	Subscript in Array							
Phase	**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.

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

Program 5.4: 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 runtimes 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.

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

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

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).

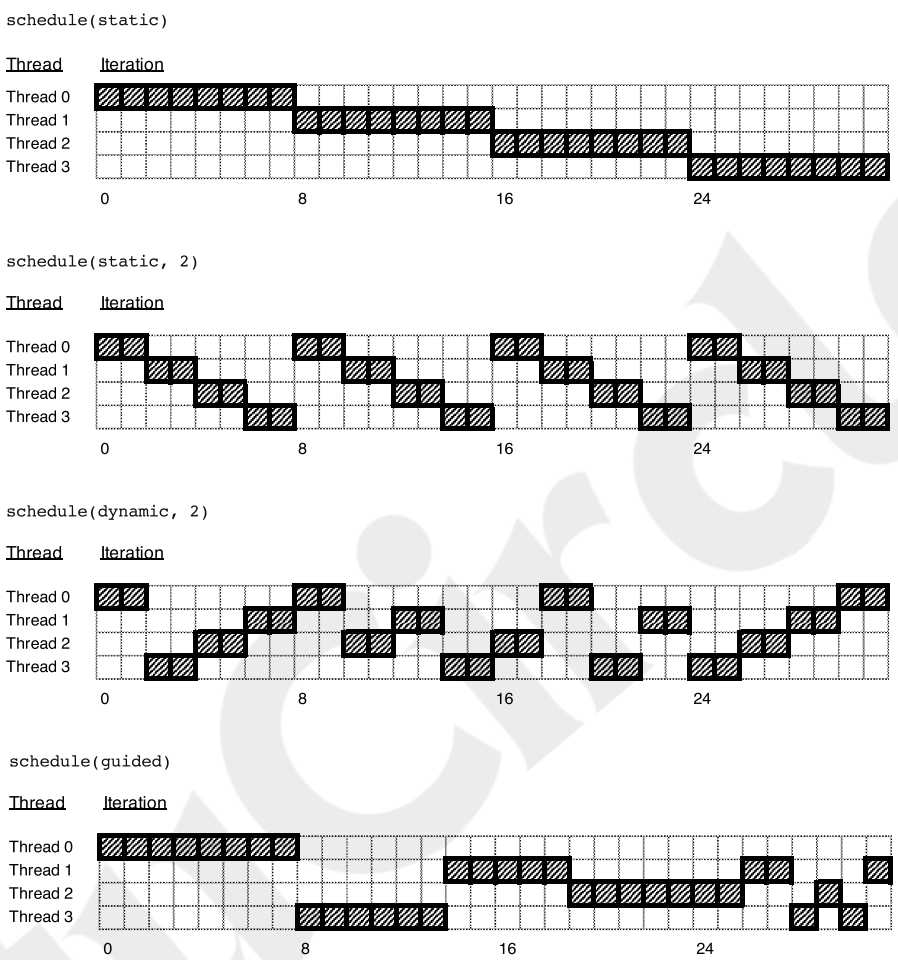

FIGURE 5.5

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, $0, 1, \ldots, 11$, 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

$$\begin{aligned} \text{Thread } 0: &\quad 0, 3, 6, 9 \\ \text{Thread } 1: &\quad 1, 4, 7, 10 \\ \text{Thread } 2: &\quad 2, 5, 8, 11 \end{aligned}$$

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

$$\begin{aligned} \text{Thread } 0: &\quad 0, 1, 6, 7 \\ \text{Thread } 1: &\quad 2, 3, 8, 9 \\ \text{Thread } 2: &\quad 4, 5, 10, 11 \end{aligned}$$

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

$$\begin{aligned} \text{Thread } 0: &\quad 0, 1, 2, 3 \\ \text{Thread } 1: &\quad 4, 5, 6, 7 \\ \text{Thread } 2: &\quad 8, 9, 10, 11 \end{aligned}$$

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

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

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

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

5.7.3 The `dynamic` and `guided` schedule types

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

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

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

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

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

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

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

5.7.4 The `runtime` schedule type

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

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

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

```
$ echo $PATH
```

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

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

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

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

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

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

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

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

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

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

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

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

5.7.5 Which schedule?

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

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

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

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

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

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

5.8 Producers and consumers

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

5.8.1 Queues

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

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

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

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

5.8.2 Message-passing

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

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

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

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

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

5.8.3 Sending messages

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

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

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

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

5.8.4 Receiving messages

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

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

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

```
queue_size = enqueued - dequeued
```

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

Thus we can implement `Try_receive` as follows:

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

5.8.5 Termination detection

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

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

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

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

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

5.8.6 Startup

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

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

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

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

```
# pragma omp barrier
```

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

5.8.7 The `atomic` directive

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

```
# pragma omp atomic
```

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

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

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

Here <op> can be one of the binary operators

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

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

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

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

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

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

5.8.8 Critical sections and locks

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.8.9 Using locks in the message-passing program

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

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

can be replaced with

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

Similarly, the code

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

can be replaced with

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

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

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

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

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

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

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

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

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

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

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

5.8.11 Some caveats

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Table 5.5 Memory and cache accesses.

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

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

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

```
my_y = x;
```

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

```
x++;
```

Finally, suppose that thread 1 now executes

```
my_z = x;
```

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

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

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

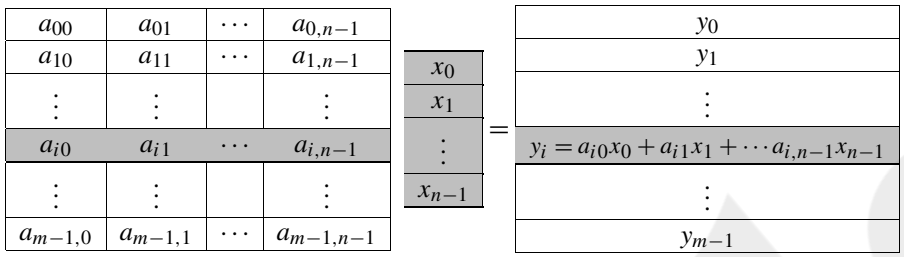

FIGURE 5.6

Matrix-vector multiplication.

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

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

See Fig. 5.6.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

and thread 3 is responsible for computing

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

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

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

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

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

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

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

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

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

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

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

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

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

5.10 Tasking

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

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

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

```
#pragma omp task
```

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

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

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

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

```
int fib(int n) {
    int i = 0;
    int j = 0;
```

```
    if (n <= 1) {
        // fibs is a global variable
        // It needs storage for n+1 ints
        fibs[n] = n;
        return n;
    }

    i = fib(n - 1);
    j = fib(n - 2);
    fibs[n] = i + j;
    return fibs[n];
}
```

This chain of recursive calls will be time-consuming, so let's execute each as a separate task that can run in parallel. We can do this by adding a `parallel` and a `single` directive before the initial (nonrecursive) call that starts `fib`, and adding **#pragma** omp task before each of the two recursive calls in `fib`. However, after we make this change, the results are incorrect—more specifically, except for `fib`[1], the sequence is all zeroes. This is because the default data scope for variables in tasks is private. So after completing each of the tasks

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

the results in `i` and `j` are lost: `i` and `j` retain their values from the initializations

```
int i = 0;
int j = 0;
```

at the beginning of the function. In other words, the memory locations that are assigned the results of `fib`(n−1) and `fib`(n−2) are not the same as the memory locations declared at the beginning of the function. So the values that are used to update `fibs`[n] are the zeroes assigned at the beginning of the function.

We can adjust the scope of `i` and `j` by declaring the variables to be `shared` in the tasks that execute the recursive call. However executing the program now will produce unpredictable results similar to our original attempt at parallelization. The problem here is that the order in which the various tasks execute isn't specified. In other words, our recursive function calls, `fib`(n − 1) and `fib`(n − 2) will be run eventually, but the thread executing the task that makes the recursive calls can continue to run and simply **return** the current value of `fibs`[n] early. We need to force this task to wait for its subtasks to complete with the `taskwait` directive, which operates as a `barrier` for tasks. We've put this all together in Program 5.6.

```
1  int fib(int n) {
2      int i = 0;
3      int j = 0;
4
5      if (n <= 1) {
6          fibs[n] = n;
7          return n;
8      }
9
10 #   pragma omp task shared(i)
11     i = fib(n - 1);
12
13 #   pragma omp task shared(j)
14     j = fib(n - 2);
15
16 #   pragma omp taskwait
17     fibs[n] = i + j;
18     return fibs[n];
19 }
```

Program 5.6: Computing the Fibonacci numbers using OpenMP tasks.

Our parallel Fibonacci program will now produce the correct results, but you may notice significant slowdowns with larger values of n; in fact, there is a good chance that the serial version of the program executes much faster! To gain an intuition as to why this occurs, recall our discussion of the overhead associated with forking and joining threads. Similarly, each task requires its own data environment to be generated upon creation, which takes time. There are a few options we can use to help reduce task creation overhead. The first option is to only create tasks in situations where n is large enough. We can do this with the **if** directive:

```
#pragma omp task shared(i) if(n > 20)
```

which in this case will restrict task creation to only occur when n is larger than 20 (chosen arbitrarily in this case based on some experimentation). Reviewing `fib` again, we can see that there will be a task executing `fib` itself, another executing `fib(n - 1)`, and a third executing `fib(n - 2)` for each recursive call. This is inefficient, because the parent task executing `fib` only launches two subtasks and then simply waits for their results. We can eliminate a task by having the parent thread perform one of the recursive calls to `fib` instead before doing the final calculation after the `taskwait` directive. On our 64-core testbed, these two changes halved the execution time of the program with $n = 35$.

While using the Tasking API requires a bit more planning and care to use—especially with data scoping and limiting runaway task creation—it allows a much broader set of problems to be parallelized by OpenMP.

5.11 Thread-safety[8]

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

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

We'll read the text into an array of strings, with one line of text per string. Then we can use a `parallel` **for** directive with a `schedule(`**static**`,1)` clause to divide the lines among the threads.

One way to tokenize a line is to use the `strtok` function in `string.h`. It has the following prototype:

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

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

Given these assumptions, we can write the `Tokenize` function shown in Program 5.7. The main function has initialized the array `lines` so that it contains the input text, and `line_count` is the number of strings stored in `lines`. Although for our purposes, we only need the `lines` argument to be an input argument, the `strtok` function modifies its input. Thus when `Tokenize` returns, `lines` will be modified. When we run the program with a single thread, it correctly tokenizes the input stream. The first time we run it with two threads and the input

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

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

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

```
1  void Tokenize(
2        char*  lines[]        /* in/out */,
3        int    line_count     /* in     */,
4        int    thread_count   /* in     */) {
5     int my_rank, i, j;
6     char *my_token;
7
8  #  pragma omp parallel num_threads(thread_count) \
9        default(none) private(my_rank, i, j, my_token) \
10       shared(lines, line_count)
11    {
12       my_rank = omp_get_thread_num();
13 #     pragma omp for schedule(static, 1)
14       for (i = 0; i < line_count; i++) {
15          printf("Thread %d > line %d = %s",
16                 my_rank, i, lines[i]);
17          j = 0;
18          my_token = strtok(lines[i], " \t\n");
19          while ( my_token != NULL ) {
20             printf("Thread %d > token %d = %s\n",
21                    my_rank, j, my_token);
22             my_token = strtok(NULL, " \t\n");
23             j++;
24          }
25       } /* for i */
26    }  /* omp parallel */
27
28 }  /* Tokenize */
```

Program 5.7: A first attempt at a multi-threaded tokenizer.

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

What happened? Recall that `strtok` caches the input line. It does this by declaring a variable to have `static` storage class. This causes the value stored in this variable

to persist from one call to the next. Unfortunately for us, this cached string is shared, not private. Thus it appears that thread 1's call to `strtok` with the second line has apparently overwritten the contents of thread 0's call with the first line. Even worse, thread 0 has found a token ("days") that should be in thread 1's output.

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

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

The "_r" is supposed to suggest that the function is *re-entrant*, which is sometimes used as a synonym for thread-safe.[9] The first two arguments have the same purpose as the arguments to `strtok`. The `saveptr` argument is used by `strtok_r` for keeping track of where the function is in the input string; it serves the purpose of the cached pointer in `strtok`. We can correct our original `Tokenize` function by replacing the calls to `strtok` with calls to `strtok_r`. We simply need to declare a **char*** variable to pass in for the third argument, and replace the calls in line 18 and line 22 with the following calls:

```
my_token = strtok_r(lines[i], " \t\n", &saveptr);
. . .
my_token = strtok_r(NULL, " \t\n", &saveptr);
```

respectively.

5.11.1 Incorrect programs can produce correct output

Notice that our original version of the tokenizer program shows an especially insidious form of program error: The first time we ran it with two threads, the program produced correct output. It wasn't until a later run that we saw an error. This, unfortunately, is not a rare occurrence in parallel programs. It's especially common in shared-memory programs. Since, for the most part, the threads are running independently of each other, as we noted back at the beginning of the chapter, the exact

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

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.