CHAPTER

17

Shared-Memory Programming

Not what we give, but what we share—
For the gift without the giver is bare;
Who gives himself with his alms feeds three—
Himself, his hungering neighbor, and me.
James Russell Lowell, *The Vision of Sir Launfal*

17.1 INTRODUCTION

In the 1980s commercial multiprocessors with a modest number of CPUs cost hundreds of thousands of dollars. Today, multiprocessors with dozens of processors are still quite expensive, but small systems are readily available for a low price. Dell, Gateway, and other companies sell dual-CPU multiprocessors for less than $5,000, and you can purchase a quad-processor system for less than $20,000.

It is possible to write parallel programs for multiprocessors using MPI, but you can often achieve better performance by using a programming language tailored for a shared-memory environment. Recently, OpenMP has emerged as a shared-memory standard. OpenMP is an application programming interface (API) for parallel programming on multiprocessors. It consists of a set of compiler directives and a library of support functions. OpenMP works in conjunction with standard Fortran, C, or C++.

This chapter introduces shared-memory parallel programming using OpenMP. You can use it in two different ways. Perhaps the only parallel computer you have access to is a multiprocessor. In that case, you may prefer to write programs using OpenMP rather than MPI.

On the other hand, you may have access to a multicomputer consisting of many nodes, each of which is a multiprocessor. This is a popular way to build large multicomputers with hundreds or thousands of processors. Consider

these examples (circa 2002):

- IBM's RS/6000 SP system contains up to 512 nodes. Each node can have up to 16 CPUs in it.
- Fujitsu's AP3000 Series supercomputer contains up to 1024 nodes, and each node consists of one or two UltraSPARC processors.
- Dell's High Performance Computing Cluster has up to 64 nodes. Each node is a multiprocessor with two Pentium III CPUs.

In this chapter you'll see how the shared-memory programming model is different from the message-passing model, and you'll learn enough OpenMP compiler directives and functions to be able to parallelize a wide variety of C code segments.

This chapter introduces a powerful set of OpenMP compiler directives:

- `parallel`, which precedes a block of code to be executed in parallel by multiple threads
- `for`, which precedes a `for` loop with independent iterations that may be divided among threads executing in parallel
- `parallel for`, a combination of the `parallel` and `for` directives
- `sections`, which precedes a series of blocks that may be executed in parallel
- `parallel sections`, a combination of the `parallel` and `sections` directives
- `critical`, which precedes a critical section
- `single`, which precedes a code block to be executed by a single thread

You'll also encounter four important OpenMP functions:

- `omp_get_num_procs`, which returns the number of CPUs in the multiprocessor on which this thread is executing
- `omp_get_num_threads`, which returns the number of threads active in the current parallel region
- `omp_get_thread_num`, which returns the thread identification number
- `omp_set_num_threads`, which allows you to fix the number of threads executing the parallel sections of code

17.2 THE SHARED-MEMORY MODEL

The shared-memory model (Figure 17.1) is an abstraction of the generic centralized multiprocessor described in Section 2.4. The underlying hardware is assumed to be a collection of processors, each with access to the same shared memory. Because they have access to the same memory locations, processors can interact and synchronize with each other through shared variables.

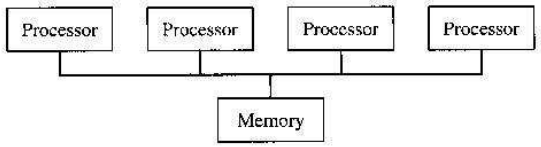

Figure 17.1 The shared-memory model of parallel computation. Processors synchronize and communicate with each other through shared variables.

The standard view of parallelism in a shared-memory program is **fork/join parallelism.** When the program begins execution, only a single thread, called the **master thread,** is active (Figure 17.2). The master thread executes the sequential portions of the algorithm. At those points where parallel operations are required, the master thread forks (creates or awakens) additional threads. The master thread and the created threads work concurrently through the parallel section. At the end of the parallel code the created threads die or are suspended, and the flow of control returns to the single master thread. This is called a join.

A key difference, then, between the shared-memory model and the message-passing model is that in the message-passing model all processes typically remain active throughout the execution of the program, whereas in the shared-memory model the number of active threads is one at the program's start and finish and may change dynamically throughout the execution of the program.

You can view a sequential program as a special case of a shared-memory parallel program: it is simply one with no fork/joins in it. Parallel shared-memory programs range from those with only a single fork/join around a single loop to those in which most of the code segments are executed in parallel. Hence the shared-memory model supports **incremental parallelization,** the process of transforming a sequential program into a parallel program one block of code at a time.

The ability of the shared-memory model to support incremental parallelization is one of its greatest advantages over the message-passing model. It allows you to profile the execution of a sequential program, sort the program blocks according to how much time they consume, consider each block in turn beginning with the most time-consuming, parallelize each block amenable to parallel execution, and stop when the effort required to achieve further performance improvements is not warranted.

Consider, in contrast, message-passing programs. They have no shared memory to hold variables, and the parallel processes are active throughout the execution of the program. Transforming a sequential program into a parallel program is not incremental at all—the chasm must be crossed with one giant leap, rather than many small steps.

In this chapter you'll encounter increasingly complicated blocks of sequential code and learn how to transform them into parallel code sections.

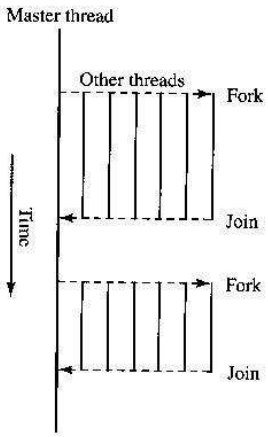

Figure 17.2 The shared-memory model is characterized by fork/join parallelism, in which parallelism comes and goes. At the beginning of execution only a single thread, called the master thread, is active. The master thread executes the serial portions of the program. It forks additional threads to help it execute parallel portions of the program. These threads are deactivated when serial execution resumes.

17.3 PARALLEL `for` LOOPS

Inherently parallel operations are often expressed in C programs as `for` loops. OpenMP makes it easy to indicate when the iterations of a `for` loop may be executed in parallel. For example, consider the following loop, which accounts for a large proportion of the execution time in our MPI implementation of the Sieve of Eratosthenes:

```
for (i = first; i < size; i += prime) marked[i] = 1;
```

Clearly there is no dependence between one iteration of the loop and another. How do we convert it into a parallel loop? In OpenMP we simply indicate to

the compiler that the iterations of a `for` loop may be executed in parallel; the compiler takes care of generating the code that forks/joins threads and **schedules** the iterations, that is, allocates iterations to threads.

17.3.1 `parallel for` Pragma

A compiler directive in C or C++ is called a **pragma.** The word *pragma* is short for "pragmatic information." A pragma is a way to communicate information to the compiler. The information is nonessential in the sense that the compiler may ignore the information and still produce a correct object program. However, the information provided by the pragma can help the compiler optimize the program.

Like other lines that provide information to the preprocessor, a pragma begins with the # character. A pragma in C or C++ has this syntax:

```
#pragma omp <rest of pragma>
```

The first pragma we are going to consider is the `parallel for` pragma. The simplest form of the `parallel for` pragma is:

```
#pragma omp parallel for
```

Putting this line immediately before the `for` loop instructs the compiler to try to parallelize the loop:

```
#pragma omp parallel for
   for (i = first; i < size; i += prime) marked[i] = 1;
```

In order for the compiler to successfully transform the sequential loop into a parallel loop, it must be able to verify that the run-time system will have the information it needs to determine the number of loop iterations when it evaluates the control clause. For this reason the control clause of the `for` loop must have **canonical shape,** as illustrated in Figure 17.3. In addition, the `for` loop must not contain statements that allow the loop to be exited prematurely. Examples include the `break` statement, `return` statement, `exit` statement, and `goto` statements

```
for (index = start; index { <, <=, >=, > } end; { index-+, ++index, index--, --index, index += inc, index -= inc, index = index + inc, index = inc + index, index = index - inc } )
```

Figure 17.3 In order to be made parallel, the control clause of a `for` loop must have canonical shape. This figure shows the legal variants. The identifiers *start, end,* and *inc* may be expressions.

to labels outside the loop. The `continue` statement is allowed, however, because its execution does not affect the number of loop iterations.

Our example `for` loop

```
for (i = first; i < size; i += prime) marked[i] = 1;
```

meets these criteria: the control clause has canonical shape, and there are no premature exits in the body of the loop. Hence the compiler can generate code that allows its iterations to execute in parallel.

During parallel execution of the `for` loop, the master thread creates additional threads, and all threads work together to cover the iterations of the loop. Every thread has its own **execution context:** an address space containing all of the variables the thread may access. The execution context includes static variables, dynamically allocated data structures in the heap, and variables on the run-time stack.

The execution context includes its own additional run-time stack, where the frames for functions it invokes are kept. Other variables may either be shared or private. A **shared variable** has the same address in the execution context of every thread. All threads have access to shared variables. A **private variable** has a different address in the execution context of every thread. A thread can access its own private variables, but cannot access the private variable of another thread.

In the case of the `parallel for` pragma, variables are by default shared, with the exception that the loop index variable is private.

Figure 17.4 illustrates shared and private variables. In this example the iterations of the `for` loop are being divided among two threads. The loop index `i` is a private variable—each thread has its own copy. The remaining variables `b` and `ptr`, as well as data allocated on the heap, are shared.

How does the run-time system know how many threads to create? The value of an environment variable called `OMP_NUM_THREADS` provides a default number of threads for parallel sections of code. In Unix you can use the `printenv` command to inspect the value of this variable and the `setenv` command to modify its value.

```
int main (int argc, char* argv[])
{
  int b[3];
  char* cptr ;
  int i;

  cptr = malloc (1);
#pragma omp parallel for
  for (i=0; i<3; i++)
   b[i]=i;
```

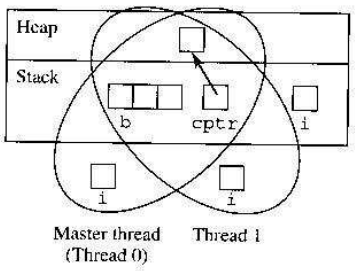

Figure 17.4 During parallel execution of the `for` loop, index `i` is a private variable, while `b`, `cptr`, and heap data are shared.

Another strategy is to set the number of threads equal to the number of multiprocessor CPUs. Let's explore the OpenMP functions that enable us to do this.

17.3.2 Function `omp_get_num_procs`

Function `omp_get_num_procs` returns the number of physical processors available for use by the parallel program. Here is the function header:

```
int omp_get_num_procs (void)
```

The integer returned by this function may be less than the total number of physical processors in the multiprocessor, depending on how the run-time system gives processes access to processors.

17.3.3 Function `omp_set_num_threads`

Function `omp_set_num_threads` uses the parameter value to set the number of threads to be active in parallel sections of code. It has this function header:

```
void omp_set_num_threads (int t)
```

Since this function may be called at multiple points in a program, you have the ability to tailor the level of parallelism to the grain size or other characteristics of the code block.

Setting the number of threads equal to the number of available CPUs is straightforward:

```
int t;
...
t = omp_get_num_procs();
omp_set_num_threads(t);
```

17.4 DECLARING PRIVATE VARIABLES

For our second example, let's look at slightly more complicated loop structure. Here is the computational heart of our MPI implementation of Floyd's algorithm:

```
for (i = 0; i < BLOCK_SIZE(id,p,n); i++)
   for (j = 0; j < n; j++)
      a[i][j] = MIN(a[i][j], a[i][k] + tmp[j]);
```

In our earlier analysis of this algorithm, we determined that either loop could be executed in parallel. Which one should we choose? If we parallelize the inner loop, then the program will fork and join threads for every iteration of the outer loop. The fork/join overhead may very well be greater than the time saved by dividing the execution of the n iterations of the inner loop among multiple threads. On the other hand, if we parallelize the outer loop, the program only incurs the fork/join overhead once.

Grain size is the number of computations performed between communication or synchronization steps. In general, increasing grain size improves the performance of a parallel program. Making the outer loop parallel results in larger grain size. It is the option we choose.

It's easy enough to direct the compiler to execute the iterations of the loop indexed by `i` in parallel. However, we need to pay attention to the variables accessed by the threads. By default, all variables are shared except loop index `i`. That makes it easy for threads to communicate with each other, but it can also cause problems.

Consider what happens when multiple threads try to execute different iterations of the `i` loop in parallel. We want every thread to work through `n` values of `j` for each iteration of the `i` loop. However, all of the threads try to initialize and increment the same shared variable `j`—meaning that there is a good chance threads will not execute all `n` iterations.

The solution is clear—we need to make `j` a private variable, too.

17.4.1 `private` Clause

A **clause** is an optional, additional component to a pragma. The `private` clause directs the compiler to make one or more variables private. It has this syntax:

```
private (<variable list>)
```

The directive tells the compiler to allocate a private copy of the variable for each thread executing the block of code the pragma precedes. In our case, we are making a `for` loop parallel. The private copies of variable `j` will be accessible only inside the `for` loop. The values are undefined on loop entry and exit.

Using the `private` clause, a correct OpenMP implementation of the doubly nested loops is

```
#pragma omp parallel for private(j)
   for (i = 0; i < BLOCK_SIZE(id,p,n); i++)
      for (j = 0; j < n; j++)
         a[i][j] = MIN(a[i][j], a[i][k] + tmp[j]);
```

Even if `j` had a previously assigned value before entering the parallel `for` loop, none of the threads can access that value. Similarly, whatever values the threads assign to `j` during execution of the parallel `for` loop, the value of the shared `j` will not be affected. Put another way, by default the value of a private variable is undefined when the parallel construct is entered, and the value is also undefined when the construct is exited.

The default condition of private variables (undefined at loop entry and exit) reduces execution time by eliminating unnecessary copying between shared variables and their private variable counterparts.

17.4.2 `firstprivate` Clause

Sometimes we want a private variable to inherit the value of the shared variable. Consider, for example, the following code segment:

```
x[0] = complex_function();
for (i = 0; i < n; i++) {
   for (j = 1 ; j < 4; j++)
      x[j] = g(i, x[j-1]);
   answer[i] = x[1] - x[3];
}
```

Assuming function `g` has no side effects, we may execute every iteration of the outer loop in parallel, as long as we make `x` a private variable. However, `x[0]` is initialized before the outer `for` loop and referenced in the first iteration of the inner loop. It is impractical to move the initialization of `x[0]` inside the outer `for` loop, because it is too time-consuming. Instead, we want each thread's private copy of array element `x[0]` to inherit the value the shared variable was assigned in the master thread.

The `firstprivate` clause, with syntax

`firstprivate (`*<variable list>*`)`

does just that. It directs the compiler to create private variables having initial values identical to the value of the variable controlled by the master thread as the loop is entered.

Here is the correct way to code the parallel loop:

```
x[0] = complex_function();
#pragma omp parallel for private(j) firstprivate(x)
for (i = 0; i < n; i++) {
   for (j = 1 ; j < 4; j++)
      x[j] = g(i, x[j-1]);
   answer[i] = x[1] - x[3];
}
```

Note that the values of the variables in the `firstprivate` list are initialized once per thread, not once per iteration. If a thread executes multiple iterations of the parallel loop and modifies the value of one of these variables in an iteration, then subsequent iterations referencing the variable will get the modified value, not the original value.

17.4.3 `lastprivate` Clause

The **sequentially last iteration** of a loop is the iteration that occurs last when the loop is executed sequentially. The `lastprivate` clause directs the compiler to generate code at the end of the parallel `for` loop that copies back to the master

thread's copy of a variable the private copy of the variable from the thread that executed the sequentially last iteration of the loop.

For example, suppose we were parallelizing the following piece of code:

```
for (i = 0; i < n; i++) {
   x[0] = 1.0;
   for (j = 1 ; j < 4; j++)
      x[j] = x[j-1] * (i+1);
   sum_of_powers[i] = x[0] + x[1] + x[2] + x[3];
}
n_cubed = x[3];
```

In the sequentially last iteration of the loop, `x[3]` gets assigned the value n^3. In order to have this value accessible outside the parallel `for` loop, we must declare `x` to be a `lastprivate` variable. Here is the correct parallel version of the loop:

```
#pragma omp parallel for private(j) lastprivate(x)
for (i = 0; i < n; i++) {
   x[0] = 1.0;
   for (j = 1 ; j < 4; j++)
      x[j] = x[j-1] * (i+1);
   sum_of_powers[i] = x[0] + x[1] + x[2] + x[3];
}
n_cubed = x[3];
```

A `parallel for` pragma may contain both `firstprivate` and `lastprivate` clauses. If the same pragma has both of these clauses, the clauses may have none, some, or all of the variables in common.

17.5 CRITICAL SECTIONS

Let's consider part of a C program that estimates the value of π using a form of numerical integration called the rectangle rule:

```
double area, pi, x;
int i, n;
...
area = 0.0;
for (i = 0; i < n; i++) {
   x = (i+0.5)/n;
   area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

Unlike the `for` loops we have already considered, the iterations of this loop are not independent of each other. Each iteration of the loop reads and updates the value of `area`. If we simply parallelize the loop:

```
   double area, pi, x;
   int i, n;
   ...
   area = 0.0;
#pragma omp parallel for private(x)
   for (i = 0; i < n; i++) {
      x = (i+0.5)/n;
      area += 4.0/(1.0 + x*x);    /* Race condition! */
   }
   pi = area / n;
```

we may not end up with the correct answer, because the execution of the assignment statement is not an atomic (indivisible) operation. This sets up a **race condition,** in which the computation exhibits nondeterministic behavior when performed by multiple threads accessing a shared variable.

See Figure 17.5. Suppose thread A and thread B are concurrently executing different iterations of the loop. Thread A reads the current value of `area` and computes the sum

```
area + 4.0/(1.0 + x*x)
```

Before it can write the value of the sum back to `area`, thread B reads the current value of `area`. Thread A updates the value of `area` with the sum. Thread B computes its sum and writes back the value. Now the value of `area` is incorrect.

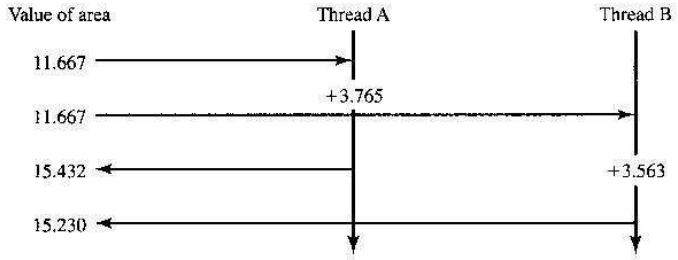

Figure 17.5 Example of a race condition. Each thread is adding a value to `area`. However, Thread B retrieves the original value of `area` before Thread A can write the new value. Hence the final value of `area` is incorrect. If Thread B had read the value of `area` after Thread A had updated it, then the final value of `area` would have been correct. In short, the absence of a critical section can lead to nondeterministic execution.

The assignment statement that reads and updates `area` must be put in a **critical section**—a portion of code that only one thread at a time may execute.

17.5.1 `critical` Pragma

We can denote a critical section in OpenMP by putting the pragma

```
#pragma omp critical
```

in front of a block of C code. (A single statement is a trivial example of a code block.) This pragma directs the compiler to enforce mutual exclusion among the threads trying to execute the block of code.

After adding the critical pragma, our code looks like this:

```
   double area, pi, x;
   int i, n;
   ...
   area = 0.0;
#pragma omp parallel for private(x)
   for (i = 0; i < n; i++) {
      x = (i+0.5)/n;
#pragma omp critical
      area += 4.0/(1.0 + x*x);
   }
   pi = area / n;
```

At this point our C/OpenMP code segment will produce the correct result. The iterations of the `for` loop are divided among the threads, and only one thread at a time may execute the assignment statement that updates the value of `area`. However, this code segment will exhibit poor speedup. Since it admits only one thread at a time, the critical section is a piece of sequential code inside the `for` loop. The time to execute this statement is nontrivial. Hence by Amdahl's Law we know the critical section will put a low ceiling on the speedup achievable by parallelizing the `for` loop.

Of course, what we are really trying to do is perform a sum-reduction of `n` values. In the next section we'll learn an efficient way to code a reduction.

17.6 REDUCTIONS

Reductions are so common that OpenMP allows us to add a reduction clause to our `parallel for` pragma. All we have to do is specify the reduction operation and the **reduction variable,** and OpenMP will take care of the details, such as storing partial sums in private variables and then adding the partial sums to the shared variable after the loop.

The reduction clause has this syntax:

`reduction(<op>:<variable>)`

where <*op*> is one of the reduction operators shown in Table 17.1 and <*variable*> is the name of the shared variable that will end up with the result of the reduction.

Here is an implementation of the π-finding code with the reduction clause replacing the critical section:

```
   double area, pi, x;
   int i, n;
   ...
   area = 0.0;
#pragma omp parallel for private(x) reduction(+:area)
   for (i = 0; i < n; i++) {
      x = (i+0.5)/n;
      area += 4.0/(1.0 + x*x);
   }
   pi = area / n;
```

Table 17.2 compares our two implementations of the rectangle rule to compute π. We set $n = 100{,}000$ and execute the programs on a Sun Enterprise Server 4000. The implementation that uses the `reduction` clause is clearly superior to the one using the `critical` pragma. It is faster when only a single thread is active, and the execution time improves when additional threads are added.

Table 17.1 OpenMP reduction operators for C and C++.

Operator	Meaning	Allowable types	Initial value
+	Sum	float, int	0
*	Product	float, int	1
&	Bitwise and	int	all bits 1
|	Bitwise or	int	0
^	Bitwise exclusive or	int	0
&&	Logical and	int	1
||	Logical or	int	0

Table 17.2 Execution times on a Sun Enterprise Server 4000 of two programs that compute π using the rectangle rule.

	Execution time of program (sec)	
Threads	Using `critical` pragma	Using `reduction` clause
1	0.0780	0.0273
2	0.1510	0.0146
3	0.3400	0.0105
4	0.3608	0.0086
5	0.4710	0.0076

17.7 PERFORMANCE IMPROVEMENTS

Sometimes transforming a sequential `for` loop into a parallel `for` loop can actually increase a program's execution time. In this section we'll look at three ways of improving the performance of parallel loops.

17.7.1 Inverting Loops

Consider the following code segment:

```
for (i = 1; i < m; i++)
   for (j = 0; j < n; j++)
      a[i][j] = 2 * a[i-1][j];
```

We can draw a data dependence diagram to help us understand the data dependences in this code. The diagram appears in Figure 17.6. We see that two rows may not be updated simultaneously, because there are data dependences between rows. However, the columns may be updated simultaneously. This means the loop indexed by `j` may be executed in parallel, but not the loop indexed by `i`.

If we insert a `parallel for` pragma before the inner loop, the resulting parallel program will execute correctly, but it may not exhibit good performance, because it will require `m-1` fork/join steps, one per iteration of the outer loop.

However, if we invert the loops:

```
#pragma parallel for private(i)
for (j = 0; j < n; j++)
   for (i = 1; i < m; i++)
      a[i][j] = 2 * a[i-1][j];
```

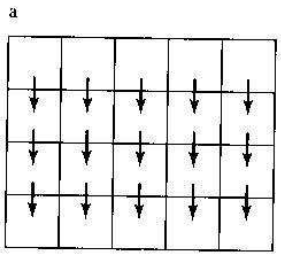

Figure 17.6 Data dependence diagram for a particular pair of nested loops shows that while columns may be updated simultaneously, rows cannot.

only a single fork/join step is required (surrounding the outer loop). The data dependences have not changed; the iterations of the loop indexed by `j` are still independent of each other. In this respect we have definitely improved the code.

However, we must always be cognizant of how code transformations affect the cache hit rate. In this case, each thread is now working through columns of `a`, rather than rows. Since C matrices are stored in row-major order, inverting loops may lower the cache hit rate, depending upon `m`, `n`, the number of active threads, and the architecture of the underlying system.

17.7.2 Conditionally Executing Loops

If a loop does not have enough iterations, the time spent forking and joining threads may exceed the time saved by dividing the loop iterations among multiple threads. Consider, for example, the parallel implementation of the rectangle rule we examined earlier:

```
   area = 0.0;
#pragma omp parallel for private(x) reduction (+:area)
   for (i = 0; i < n; i++) {
      x = (i + 0.5)/n;
      area += 4.0 / (1.0 + x * x);
   }
   pi = area / n;
```

Table 17.3 reveals the average execution time of this program segment on a Sun Enterprise Server 4000, for various values of `n` and various numbers of threads. As you can see, when `n` is 100, the sequential execution time is so small that adding threads only increases overall execution time. When `n` is 100,000, the parallel program executing on four threads achieves a speedup of 3.16 over the sequential program.

The `if` clause gives us the ability to direct the compiler to insert code that determines at run-time whether the loop should be executed in parallel or

Table 17.3 Execution time on a Sun Enterprise Server 4000 of a parallel C program that computes π using the rectangle rule, as a function of number of rectangles and number of threads.

	Execution time (msec)	
Threads	$n = 100$	$n = 100{,}000$
1	0.964	27.288
2	1.436	14.598
3	1.732	10.506
4	1.990	8.648

sequentially. The clause has this syntax:

`if (`*<scalar expression>*`)`

If the scalar expression evaluates to true, the loop will be executed in parallel. Otherwise, it will be executed serially.

For example, here is how we could add an `if` clause to the `parallel for` pragma in the parallel program computing π using the rectangle rule:

```
#pragma omp parallel for private(x) reduction(+:area) if(n > 5000)
   for (i = 0; i < n; i++) {
   ...
```

In this case loop iterations will be divided among multiple threads only if $n > 5{,}000$.

17.7.3 Scheduling Loops

In some loops the time needed to execute different loop iterations varies considerably. For example, consider the following doubly nested loop that initializes an upper triangular matrix:

```
for (i = 0; i < n; i++)
   for (j = i; j < n; j++)
      a[i][j] = alpha_omega(i,j);
```

Assuming there are no data dependences among iterations, we would prefer to execute the outermost loop in parallel in order to minimize fork/join overhead. If every call to function `alpha_omega` takes the same amount of time, then the first iteration of the outermost loop (when `i` equals 0) requires `n` times more work than the last iteration (when `i` equals `n-1`). Inverting the two loops will not remedy the imbalance.

Suppose these n iterations are being executed on t threads. If each thread is assigned a contiguous block of either $\lceil n/t \rceil$ or $\lfloor n/t \rfloor$ threads, the parallel loop execution will have poor efficiency, because some threads will complete their share of the iterations much faster than others.

The `schedule` clause allows us to specify how the iterations of a loop should be scheduled, that is, allocated to threads. In a **static schedule,** all iterations are allocated to threads before they execute any loop iterations. In a **dynamic schedule,** only some of the iterations are allocated to threads at the beginning of the loop's execution. Threads that complete their iterations are then eligible to get additional work. The allocation process continues until all of the iterations have been distributed to threads. Static schedules have low overhead but may exhibit high load imbalance. Dynamic schedules have higher overhead but can reduce load imbalance.

In both static and dynamic schedules, contiguous ranges of iterations called **chunks** are assigned to threads. Increasing the chunk size can reduce overhead

and increase the cache hit rate. Reducing the chunk size can allow finer balancing of workloads.

The schedule clause has this syntax:

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

In other words, the schedule type is required, but the chunk size is optional. With these two parameters it's easy to describe a wide variety of schedules:

- `schedule(static)`: A static allocation of about `n/t` contiguous iterations to each thread.
- `schedule(static,C)`: An interleaved allocation of chunks to tasks. Each chunk contains C contiguous iterations.
- `schedule(dynamic)`: Iterations are dynamically allocated, one at a time, to threads.
- `schedule(dynamic,C)`: A dynamic allocation of C iterations at a time to the tasks.
- `schedule(guided,C)`: A dynamic allocation of iterations to tasks using the guided self-scheduling heuristic. **Guided self-scheduling** begins by allocating a large chunk size to each task and responds to further requests for chunks by allocating chunks of decreasing size. The size of the chunks decreases exponentially to a minimum chunk size of C.
- `schedule(guided)`: Guided self-scheduling with a minimum chunk size of 1.
- `schedule(runtime)`: The schedule type is chosen at run-time based on the value of the environment variable `OMP_SCHEDULE`. For example, the Unix command

  ```
  setenv OMP_SCHEDULE "static,1"
  ```

  would set the run-time schedule to be an interleaved allocation.

When the `schedule` clause is not included in the `parallel for` pragma, most run-time systems default to a simple static scheduling of consecutive loop iterations to tasks.

Going back to our original example, the run-time of any particular iteration of the outermost `for` loop is predictable. An interleaved allocation of loop iterations balances the workload of the threads:

```
#pragma omp parallel for private(j) schedule(static,1)
for (i = 0; i < n; i++)
   for (j = i; j < n; j++)
      a[i][j] = alpha_omega(i,j);
```

Increasing the chunk size from 1 could improve the cache hit rate at the expense of increasing the load imbalance. The best value for the chunk size is system-dependent.

17.8 MORE GENERAL DATA PARALLELISM

To this point we have focused on the parallelization of simple `for` loops. They are perhaps the most common opportunity for parallelism, particularly in programs that have already been written in MPI. However, we should not ignore other opportunities for concurrency. In this section we look at two examples of data parallelism outside simple `for` loops.

First let's consider an algorithm to process a linked list of tasks. We considered a similar algorithm when we designed a solution to the document classification problem in Chapter 9. In that design, we assumed a message-passing model. Because that model has no shared memory, we gave a single process, which we called the manager, responsibility for maintaining the entire list of tasks. Worker tasks sent messages to the manager when they were ready to process another task.

In contrast, the shared-memory model allows every thread to access the same "to-do" list, so there is no need for a separate manager thread.

The following code segments are part of a program that processes work stored in a singly linked to-do list (see Figure 17.7):

```
int main (int argc, char argv[])
{
   struct job_struct job_ptr;
   struct task_struct task_ptr;

   ...
   task_ptr = get_next_task (&job_ptr);
   while (task_ptr != NULL) {
      complete_task (task_ptr);
      task_ptr = get_next_task (&job_ptr);
   }
   ...
}

char get_next_task(struct job_struct job_ptr) {
   struct task_struct answer;
   if (job_ptr == NULL) answer = NULL;
   else {
      answer = (job_ptr)->task;
      job_ptr = (job_ptr)->next;
   }
   return answer;
}
```

How would we like this algorithm to execute in parallel? We want every thread to do the same thing: repeatedly take the next task from the list and complete it, until there are no more tasks to do. We need to ensure that no two threads take the same task from the list. In other words, it is important to execute function `get_next_task` atomically.

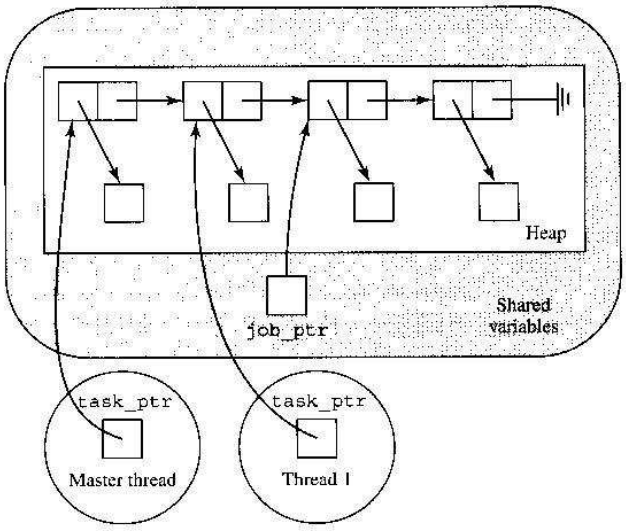

Figure 17.7 Two threads work their way through a singly linked "to do" list. Variable `job_ptr` must be shared, while `task_ptr` must be a private variable.

17.8.1 `parallel` Pragma

The parallel pragma precedes a block of code that should be executed by all of the threads. It has this syntax:

```
#pragma omp parallel
```

If the code we want executed in parallel is not a simple statement (such as an assignment statement, `if` statement, or `for` loop) we can use curly braces to create a block of code from a statement group.

Note that unlike the `parallel for` pragma, which divided the iterations of the `for` loop among the active threads, the execution of the code block after the parallel program is replicated among the threads. Our section of function `main` now looks like this:

```
int main (int argc, char argv[])
{
   struct job_struct job_ptr;
   struct task_struct task_ptr;

#pragma omp parallel private (task_ptr)
   {
      task_ptr = get_next_task (&job_ptr);
```

```
      while (task_ptr != NULL) {
         complete_task (task_ptr);
         task_ptr = get_next_task (&job_ptr);
      }
   }
}
```

Now we need to ensure function `get_next_task` executes atomically. Otherwise, allowing two threads to execute function `get_next_task` simultaneously may result in more than one thread returning from the function with the same value of `task_ptr`.

We use the `critical` pragma to ensure mutually exclusive execution of this critical section of code. Here is the rewritten function `get_next_task`:

```
char get_next_task(struct job_struct job_ptr) {
   struct task_struct answer;

#pragma omp critical
   {
      if (job_ptr == NULL) answer = NULL;
      else {
         answer = (job_ptr)->task;
         job_ptr = (job_ptr)->next;
      }
   }
   return answer;
}
```

17.8.2 Function `omp_get_thread_num`

Earlier in this chapter we computed π using the rectangle rule. In Chapter 10 we computed π using the Monte Carlo method. The idea, illustrated in Figure 17.8, is to generate pairs of points in the unit square (where each coordinate varies between 0 and 1). We count the fraction of points inside the circle (those points for which $x^2 + y^2 \leq 1$). The expected value of this fraction is $\pi/4$; hence multiplying the fraction by 4 gives an estimate of π.

Here is the C code implementing the algorithm:

```
int      count;          /* Points inside unit circle */
unsigned short xi[3];    /* Random number seed */
int      i;
int      samples;        /* Points to generate */
double   x, y;           /* Coordinates of point */
samples = atoi (argv[1]);
xi[0] = atoi (argv[2]);
xi[1] = atoi (argv[3]);
```

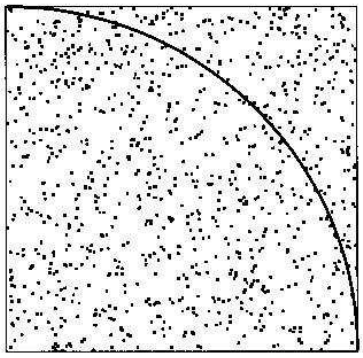

Figure 17.8 Example of a Monte Carlo algorithm to compute π. In this example we have generated 1,000 pairs from a uniform distribution between 0 and 1. Since 773 pairs are inside the unit circle, our estimate of π is 4(773/1000), or 3.092.

```
xi[2] = atoi (argv[4]);
count = 0;
for (i = 0; i < samples; i++) {
   x = erand48(xi);
   y = erand48(xi);
   if (x*x+y*y <= 1.0) count++;
}
printf ("Estimate of pi: %7.5f\n", 4.0*count/samples);
```

If we want to speed the execution of the program by using multiple threads, we must ensure that each thread is generating a different stream of random numbers. Otherwise, each thread would generate the same sequence of (x, y) pairs, and there would be no increase in the precision of the answer through the use of parallelism. Hence `xi` must be a private variable, and must find some way for each thread to initialize array `xi` with unique values. That means we need to have some way of distinguishing threads.

In OpenMP every thread on a multiprocessor has a unique identification number. We can retrieve this number using the function `omp_get_thread_num`, which has this header:

```
int omp_get_thread_num (void)
```

If there are t active threads, the thread identification numbers are integers ranging from 0 through $t - 1$. The master thread always has identification number 0.

Assigning the thread identification number to `xi[2]` ensures each thread has a different random number seed.

17.8.3 Function `omp_get_num_threads`

In order to divide the iterations among the threads, we must know the number of active threads. Function `omp_get_num_threads`, with this header

```
int omp_get_num_threads(void)
```

returns the number of threads active in the current parallel region. We can use this information, as well as the thread identification number, to divide the iterations among the threads.

Each thread will accumulate its count of points inside the circle in a private variable. When each thread completes the `for` loop, it will add its subtotal to `count` inside a critical section.

The OpenMP implementation of the Monte Carlo π-finding algorithm appears in Figure 17.9.

17.8.4 `for` Pragma

The `parallel` pragma can also come in handy when parallelizing `for` loops. Consider this doubly nested loop:

```
for (i = 0; i < m; i++) {
   low = a[i];
   high = b[i];
   if (low > high) {
      printf ("Exiting during iteration %d\n", i);
      break;
   }
   for (j = low; j < high; j++)
      c[j] = (c[j] - a[i])/b[i];
}
```

We cannot execute the iterations of the outer loop in parallel, because it contains a `break` statement. If we put a `parallel for` pragma before the loop indexed by `j`, there will be a fork/join step for every iteration of the outer loop. We would like to avoid this overhead. Previously, we showed how inverting `for` loops could solve this problem, but that approach doesn't work here because of the data dependences.

If we put the `parallel` pragma immediately in front of the loop indexed by `i`, then we'll only have a single fork/join. The default behavior is that *every* thread executes *all* of the code inside the block. Of course, we want the threads to divide up the iterations of the inner loop. The `for` pragma directs the compiler to do just that:

```
#pragma omp for
```

```
/*
 * OpenMP implementation of Monte Carlo pi-finding algorithm
 */

#include <stdio.h>
int main (int argc, char *argv[]){
   int       count;         /* Points inside unit circle */
   int       i;
   int       local_count;   /* This thread's subtotal */
   int       samples;       /* Points to generate */
   unsigned short xi[3];    /* Random number seed */
   int       t;             /* Number of threads */
   int       tid;           /* Thread id */
   double    x, y;          /* Coordinates of point */

   /* Number of points and number of threads are
      command-line arguments */

   samples = atoi(argv[1]);
   omp_set_num_threads (atoi(argv[2]));

   count = 0;
#pragma omp parallel private(xi,t,i,x,y,local_count)
   {
      local_count = 0;
      xi[0]   atoi(argv[3]);
      xi[1] = atoi(argv[4]);
      xi[2] = tid = omp_get_thread_num();
      t = omp_get_num_threads();

      for (i = tid; i < samples; i += t) {
         x = erand48(xi);
         y = erand48(xi);
         if (x*x+y*y <= 1.0) local_count++;
      }
#pragma omp critical
      count += local_count;
   }
   printf ("Estimate of pi: %7.5f\n", 4.0*count/samples);
}
```

Figure 17.9 This C/OpenMP program uses the Monte Carlo method to compute π.

With these pragmas added, our code segment looks like this:

```
#pragma omp parallel private(i,j)
for (i = 0; i < m; i++) {
   low = a[i];
   high = b[i];
   if (low > high) {
      printf ("Exiting during iteration %d\n", i);
      break;
   }
```

```
#pragma omp for
   for (j = low; j < high; j++)
      c[j] = (c[j] - a[i])/b[i];
}
```

Our work is not yet complete, however.

17.8.5 `single` Pragma

We have parallelized the execution of the loop indexed by `j`. What about the other code inside the outer loop? We certainly don't want to see the error message more than once.

The `single` pragma tells the compiler that only a single thread should execute the block of code the pragma precedes. Its syntax is:

```
#pragma omp single
```

Adding the `single` pragma to the code block, we now have:

```
#pragma omp parallel private(i,j)
for (i = 0; i < m; i++) {
   low = a[i];
   high = b[i];
   if (low > high) {
#pragma omp single
      printf ("Exiting during iteration %d\n", i);
      break;
   }
#pragma omp for
   for (j = low; j < high; j++)
      c[j] = (c[j] - a[i])/b[i];
}
```

The code block now executes correctly, but we can improve its performance.

17.8.6 `nowait` Clause

The compiler puts a barrier synchronization at the end of every `parallel for` statement. In the example we have been considering, this barrier is necessary, because we need to ensure that every thread has completed one iteration of the loop indexed by `i` before any thread begins the next iteration. Otherwise, a thread might change the value of `low` or `high`, altering the number of iterations of the `j` loop performed by another thread.

On the other hand, if we make `low` and `high` private variables, there is no need for the barrier at the end of the loop indexed by `j`. The `nowait` clause, added to a `parallel for` pragma, tells the compiler to omit the barrier synchronization at the end of the parallel `for` loop.

After making low and high private and adding the nowait clause, our final version of our example code segment is:

```
#pragma omp parallel private(i,j,low,high)
for (i = 0; i < m; i++) {
   low = a[i];
   high = b[i];
   if (low > high) {
#pragma omp single
      printf ("Exiting during iteration %d\n", i);
      break;
   }
#pragma omp for nowait
   for (j = low; j < high; j++)
      c[j] = (c[j] - a[i])/b[i];
}
```

17.9 FUNCTIONAL PARALLELISM

To this point we have focused entirely on exploiting data parallelism. Another source of concurrency is functional parallelism. OpenMP allows us to assign different threads to different portions of code.

Consider, for example, the following code segment:

```
v = alpha();
w = beta();
x = gamma(v, w);
y = delta();
printf ("%6.2f\n", epsilon(x,y));
```

If all of the functions are side-effect free, we can represent the data dependences as shown in Figure 17.10. Clearly functions alpha, beta, and delta may be executed in parallel. If we execute these functions concurrently, there is no more

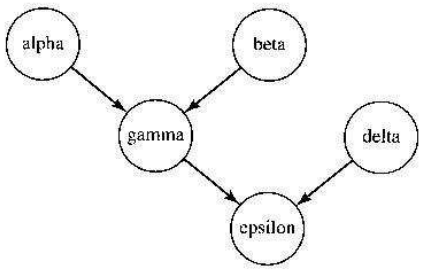

Figure 17.10 Data dependence diagram for code segment of Section 17.9.

functional parallelism to exploit, because function `gamma` must be called after functions `alpha` and `beta` and before function `epsilon`.

17.9.1 `parallel sections` Pragma

The `parallel sections` pragma precedes a block of k blocks of code that may be executed concurrently by k threads. It has this syntax:

```
#pragma omp parallel sections
```

17.9.2 `section` Pragma

The `section` pragma precedes each block of code within the encompassing block preceded by the `parallel sections` pragma. (The `section` pragma may be omitted for the first parallel section after the `parallel sections` pragma.)

In the example we considered, the calls to functions `alpha`, `beta`, and `delta` could be evaluated concurrently. In our parallelization of this code segment, we use curly braces to create a block of code containing these three assignment statements. (Recall that an assignment statement is a trivial example of a code block. Hence a block containing three assignment statements is a block of three blocks of code.)

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

Note that we reordered the assignment statements to bring together the three that could be executed in parallel.

17.9.3 `sections` Pragma

Let's take another look at the data dependence diagram of Figure 17.10. There is a second way to exploit functional parallelism in this code segment. As we noted earlier, if we execute functions `alpha`, `beta`, and `delta` in parallel, there are no further opportunities for functional parallelism. However, if we execute only functions `alpha` and `beta` in parallel, then after they return we may execute functions `gamma` and `delta` in parallel.

In this design we have two different parallel sections, one following the other. We can reduce fork/join costs by putting all four assignment statements in a single

block preceded by the `parallel` pragma, then using the `sections` pragma to identify the first and second pairs of functions that may execute in parallel.

The `sections` pragma with syntax

```
#pragma omp sections
```

appears inside a `parallel` block of code. It has exactly the same meaning as the `parallel sections` pragma we have already described.

Here is another way to express functional parallelism in the code segment we have been considering, using the `sections` pragma:

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

In one respect this solution is better than the first one we presented, because it has two parallel sections of code, each requiring two threads. Our first solution has only a single parallel section of code that required three threads. If only two processors are available, the second section of code could result in higher efficiency. Whether or not that is the case depends upon the execution times of the individual functions.

17.10 SUMMARY

OpenMP is an API for shared-memory parallel programming. The shared-memory model relies upon fork/join parallelism. You can envision the execution of a shared-memory program as periods of sequential execution alternating with periods of parallel execution. A master thread executes all of the sequential code. When it reaches a parallel code segment, it forks other threads. The threads communicate with each other via shared variables. At the end of the parallel code segment, these threads synchronize, rejoining the master thread.

This chapter has introduced OpenMP pragmas and clauses that can be used to transform a sequential C program into one that runs in parallel on a multiprocessor. First we considered the parallelization of `for` loops. In C programs data parallelism is often expressed in the form of `for` loops. We use the `parallel for` pragma to indicate to the compiler those loops whose iterations may be performed in parallel. There are certain restrictions on `for` loops that may be executed in parallel. The control clause must be simple, so that the run-time system can determine, before the loop executes, how many iterations it will have. The loop cannot have a `break` statement, `goto` statement, or another statement that allows early loop termination.

We also discussed how to take advantage of functional parallelism through the use of the `parallel sections` pragma. This pragma precedes a block of blocks of code, where each of the inner blocks, or sections, represents an independent task that may be performed in parallel with the other sections.

The `parallel` pragma precedes a block of code that should be executed in parallel by all threads. When all threads execute the same code, the result is SPMD-style parallel execution similar to that exhibited by many of our programs using MPI. A `for` pragma or a `sections` pragma may appear inside the block of code marked with a `parallel` pragma, allowing the compiler to exploit data or functional parallelism.

We also use pragmas to point out areas within parallel sections that must be executed sequentially. The `critical` pragma indicates a block of code forming a critical section where mutual exclusion must be enforced. The `single` pragma indicates a block of code that should only be executed by one of the threads.

We can convey additional information to the compiler by adding clauses to pragmas. The `private` clause gives each thread its own copy of the listed variables. Values can be copies between the original variable and private variables using the `firstprivate` and/or the `lastprivate` clauses. The `reduction` clause allows the compiler to generate efficient code for reduction operations happening inside a parallel loop. The `schedule` clause lets you specify the way loop iterations are allocated to tasks. The `if` clause allows the system to determine at run-time if a construct should be executed sequentially or by multiple threads. The `nowait` clause eliminates the barrier synchronization at the end of the parallel construct.

While we have introduced clauses in the context of particular pragmas, most clauses can be applied to most pragmas. Table 17.4 lists which of the clauses we have introduced in this chapter may be attached to which pragmas.

We have examined various ways in which the performance of parallel `for` loops can be enhanced. The strategies are inverting loops, conditionally parallelizing loops, and changing the way in which loop iterations are scheduled.

Table 17.5 compares OpenMP with MPI. Both programming environments can be used to program multiprocessors. MPI is suitable for programming multicomputers. Since OpenMP has shared variables, OpenMP is not appropriate for generic multicomputers in which there is no shared memory. MPI also makes it easier for the programmer to take control of the memory hierarchy. On the

Table 17.4 This table summarizes which clauses may be attached to which pragmas.

Pragma	Clauses allowed
`critical`	*None*
`for`	`firstprivate, lastprivate, nowait, private, reduction, schedule`
`parallel`	`firstprivate, if, lastprivate, private, reduction`
`parallel for`	`firstprivate, if, lastprivate, private, reduction, schedule`
`parallel sections`	`firstprivate, if, lastprivate, private, reduction`
`sections`	`firstprivate, lastprivate, nowait, private, reduction`
`single`	`firstprivate, nowait, private`

Note: OpenMP has additional clauses not introduced in this chapter.

Table 17.5 Comparison of OpenMP and MPI.

Characteristic	OpenMP	MPI
Suitable for multiprocessors	Yes	Yes
Suitable for multicomputers	No	Yes
Supports incremental parallelization	Yes	No
Minimal extra code	Yes	No
Explicit control of memory hierarchy	No	Yes

other hand, OpenMP has the significant advantage of allowing programs to be incrementally parallelized. In addition, unlike programs using MPI, which often are much longer than their sequential counterparts, programs using OpenMP are usually not much longer than the sequential codes they displace.

17.11 KEY TERMS

canonical shape
chunk
clause
critical section
dynamic schedule
execution context
fork/join parallelism
grain size
guided self-scheduling
incremental parallelization
master thread
pragma
`private` clause
private variable
race condition
reduction variable
schedule
sequentially last iteration
shared variable
static schedule

17.12 BIBLIOGRAPHIC NOTES

The URL for the official OpenMP Web site is `www.OpenMP.org`. You can download the official OpenMP specifications for the C/C++ and Fortran versions of OpenMP from this site.

Parallel Programming in OpenMP by Chandra et al. is an excellent introduction to this shared-memory application programming interface [16]. It provides broader and deeper coverage of the features of OpenMP. It also discusses performance tuning of OpenMP codes.

17.13 EXERCISES

17.1 Of the four OpenMP functions presented in this chapter, which two have the closest analogs to MPI functions? Name the MPI function each of these functions is similar to.

17.2 For each of the following code segments, use OpenMP pragmas to make the loop parallel, or explain why the code segment is not suitable for parallel execution.

a.
```
for (i = 0; i < (int) sqrt(x); i++) {
   a[i] = 2.3 * i;
   if (i < 10) b[i] = a[i];
}
```

b.
```
flag = 0;
for (i = 0; (i < n) & (!flag); i++) {
   a[i] = 2.3 * i;
   if (a[i] < b[i]) flag = 1;
}
```

c.
```
for (i = 0; i < n; i++)
   a[i] = foo(i);
```

d.
```
for (i = 0; i < n; i++) {
   a[i] = foo(i);
   if (a[i] < b[i]) a[i] = b[i];
}
```

e.
```
for (i = 0; i < n; i++) {
   a[i] = foo(i);
   if (a[i] < b[i]) break;
}
```

f.
```
dotp = 0;
for (i = 0; i < n; i++)
   dotp += a[i] * b[i];
```

g.
```
for (i = k; i < 2*k; i++)
   a[i] = a[i] + a[i-k];
```

h.
```
for (i = k; i < n; i++)
   a[i] = b * a[i-k];
```

17.3 Suppose OpenMP did not have the `reduction` clause. Show how to implement an efficient parallel reduction by adding a private variable and using the `critical` pragma. Illustrate your methodology using the π-estimation program segment from Section 17.5.

17.4 Section 17.7.3 discusses an interleaved scheduling of tasks to balance workloads among threads initializing an upper triangular matrix. Explain why increasing the chunk size from 1 could improve the cache hit rate.

17.5 Give an example of a simple parallel `for` loop that would probably execute faster with an interleaved static scheduling than by giving each task a single contiguous chunk of iterations. Your example should not have nested loops.

17.6 Give an original example of a parallel `for` loop that would probably execute in less time if it were dynamically scheduled rather than statically scheduled.

17.7 In Section 17.8 we develop a parallel code segment allowing multiple threads to work through a single "to-do" list. Explain how two threads could end up processing the same task if function `get_next_task` is not executed atomically.

17.8 Figure 17.9 illustrates a C/OpenMP program that uses the Monte Carlo algorithm to compute π. Note that the iterations of the `for` loop are divided among the threads explicitly. Implement another version of this program that uses the `for` pragma to delegate the allocation of loop iterations to the run-time system. Benchmark your program for various values of n (number of samples) and t (number of threads).

17.9 Use OpenMP directives to express as much parallelism as possible in the following code segment from Winograd's matrix multiplication algorithm (adapted from Baase and Van Gelder [5]).

```
for (i = 0; i < m; i++) {
   rowterm[i] = 0.0;
   for (j = 0; j < p; j++)
      rowterm[i] += a[i][2*j] * a[i][2*j+1];
}
for (i = 0; i < q; i++) {
   colterm[i] = 0.0;
   for (j = 0; j < p; j++)
      colterm[i] += b[2*j][i] * b[2*j+1][i];
}
```

17.10 Use OpenMP directives to implement a parallel program for the Sieve of Eratosthenes. Benchmark your program for various values of n and t (number of threads).

17.11 Use OpenMP directives to implement a parallel program for Floyd's algorithm (Figure 6.2). Benchmark your program for different values of n and t (number of threads).

17.12 Use OpenMP directives to implement a parallel version of matrix-vector multiplication (Figure 8.1). Benchmark your program for different values of n and t (number of threads).

17.13 Use OpenMP directives to implement a parallel program that solves a dense system of linear equations using Gaussian elimination with row pivoting, followed by back substitution (Figure 12.7). Benchmark your program for different values of n and t (number of threads).

17.14 Use OpenMP directives to implement a parallel version of the conjugate gradient method (Figure 12.13), assuming the coefficient matrix A is symmetrically banded. Benchmark your program for different values of n and t (number of threads).

17.15 Use OpenMP directives to implement a parallel version of Parallel Sorting by Regular Sampling (Figure 14.5). Benchmark your program for different values of n and t (number of threads).

17.16 Use OpenMP directives to implement a parallel program that solves the 15-puzzle (Chapter 17). For a variety of scrambled puzzles, benchmark your program for different values of t (number of threads).