**UNIT III SHARED MEMORY PROGRAMMING WITH OpenMP 9**

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

**3.1 Introduction to OpenMP**

1. OpenMP is an API for shared-memory parallel programming. The “MP” in OpenMP stands for “multiprocessing,” a term that is synonymous with shared-memory parallel computing.
2. 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 cores or CPUs, all of which have access to main memory.
3. 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.
4. 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.
5. 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.
6. 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.

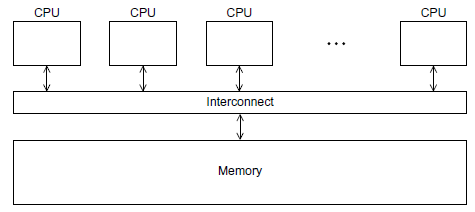
****

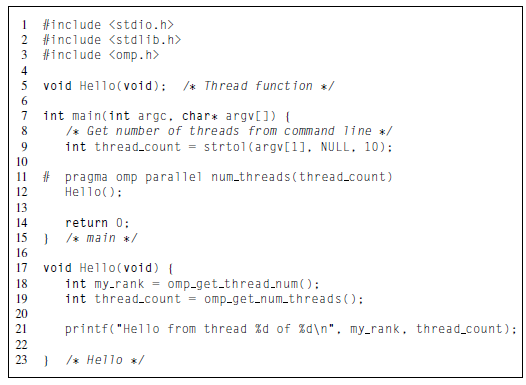
Fig. 3.1 A shared-memory system

Let’s take a look at a very simple example, a “hello, world” program that uses OpenMP.

**3.2. OpenMP Execution Model**

**Getting Started**

1. OpenMP provides what’s known as a “directives-based” shared-memory API. In C and C++, this means that there are special preprocessor instructions known as **pragmas**. Pragmas are typically added to a system to allow behaviors that aren’t part of the basic C speciﬁcation.
2. Pragmas (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.



Program 3.1: A “hello,world” program that uses OpenMP

**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

**$ ./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’snoguaranteethattheoutputwillappearinthread-rankorder.Forexample,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

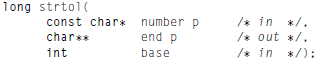
**$ ./omp\_hello 1**

and we would get the output

**Hello from thread 0 of 1**

**The Program – omp\_helloc.c**

1. 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.
2. 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



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

***pragma omp parallel* directive**

1. We’ve already seen that pragmas in C and C++ start with

**# pragma**

OpenMP pragmas always begin with

**# pragma omp**

1. 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.
2. Recollect 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.
3. At its most basic the parallel directive is simply

**# pragma omp parallel**

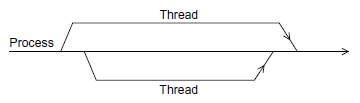


Fig.3.2 A process forking and joining two threads

we’ll usually specify the number of threads on the command line, so we’ll modify our parallel directives with the num\_threads clause.

***num\_threads* clause**

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)**

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 additionalthreads are started. In OpenMP parlance, the collection of threads executing the parallel block—the original thread and the new threads—is called a **team**, the original thread is called the **master**, and the additional threads are called **slaves**. Each thread in the team executes the block following the directive, so in 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.

**omp\_get\_thread\_num()** and **omp\_get\_num\_threads()**

Since each thread has its own stack, a thread executing the Hello function will create its own private, local variables in the function. In our example, when the function is called, each thread will get its rank or id and the number of threads in the team by calling the OpenMP functions **omp\_get\_thread\_num** and **omp\_get\_num\_threads**, respectively. The rank or id of a thread is an int that is in the range 0,1,..., thread count −1. The syntax for these functions is

**int omp\_get\_thread\_num(void);**

**int omp\_get\_num\_threads(void);**

**Error checking**

1. 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\_ge\_num\_threads will cause errors.
2. To handle these problems, we can check whether the preprocessor macro \_OPENMP is deﬁned. If this is deﬁned, we can include omp.h and make the calls to the OpenMP functions. We might make the following modiﬁcations to our program.
3. Instead of simply including omp.h in the line #include <omp.h>. we can check for the deﬁnition of \_OPENMP before trying to include it:

**#ifdef \_OPENMP**

**# include <omp.h>**

**#endif**

**3.3 The Shared-Memory Model**

1. The shared-memory model (Fig 3.3) is an abstraction of the generic centralized multiprocessor described in previous section. 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.

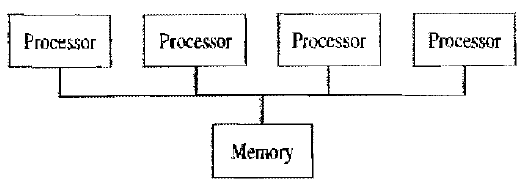


Figure 3.3 The shared memory model to parallel computation. Processors synchronize and communicate with each other through shared variables.

1. 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 (Fig 3.4). 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**.
2. 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.
3. 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 lime.

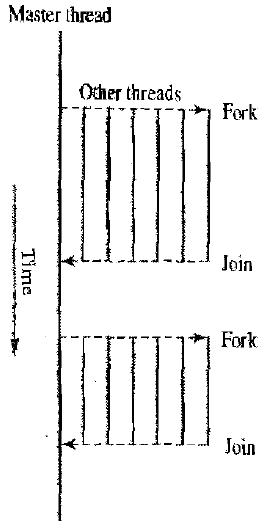


Fig 3.4 The shared-memory model is characterized by fork/join in parallelism, in which parallelism comes and goes. At the beginning of execution only a single thread, called the master thread, is active. The master thread executes the serial portions of the program. It forks additional threads to help it execute parallel portions of the program. These threads are deactivated when serial execution resumes.

1. 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 ill 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.

**3.4 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.

If the endpoints of the subinterval are xi and xi+1, then the length of the subinterval is **h=xi+1−xi.** Also, if the lengths of the two vertical segments are f(xi) and f(xi+1), then the area of the trapezoid is

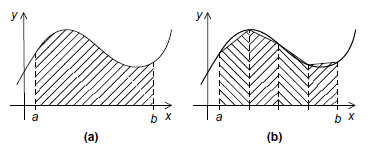
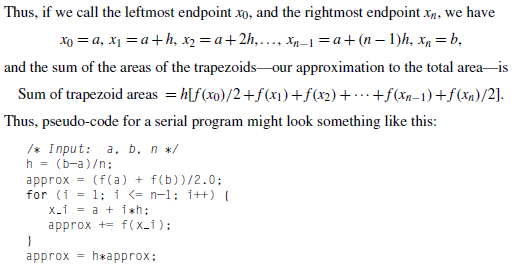


Fig. 3.5 The trapezoidal rule: (a) area to be estimated and (b) approximate area using trapezoids

Since we chose the n subintervals so that they would all have the same length, we also know that if the vertical lines bounding the region are x=a and x=b, then





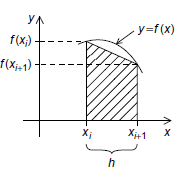


Fig 3.6 One trapezoid

**A first OpenMP version**

It is not the most attractive word, but, as we noted in Chapter 1, people who write parallel programs do use the verb “parallelize” to describe the process of converting a serial program or algorithm into a parallel program. Recall that we can design a parallel program using four basic steps:

1. Partition the problem solution into tasks.

2. Identify the communication channels between the tasks.

3. Aggregate the tasks into composite tasks.

4. Map the composite tasks to cores.

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).

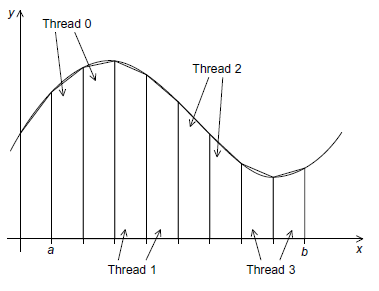
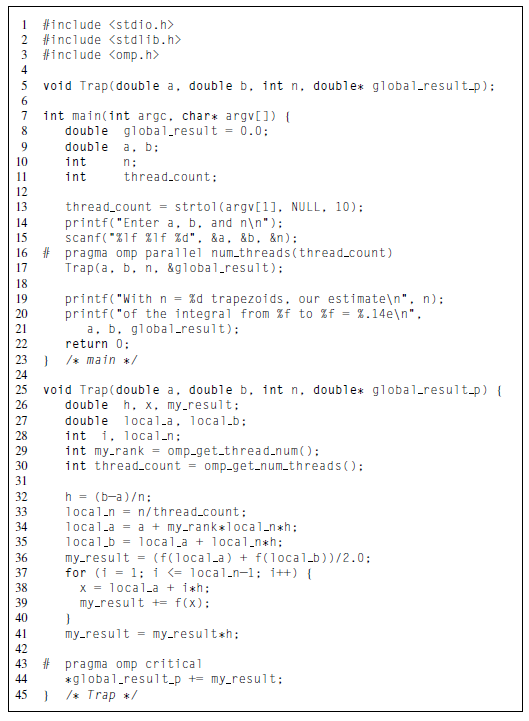


Fig 3.7 Assignment of trapezoids to threads

****

Program 3.2: First OpenMP trapezoidal rule program

1. We identiﬁed two types of tasks:

a. Computation of the areas of individual trapezoids, and

b. Adding the areas of trapezoids.

1. There is no communication among the tasks in the ﬁrst collection, but each task in the ﬁrst collection communicates with task 1(b).
2. We assumed that there would be many more trapezoids than cores, so we aggregated tasks by assigning a contiguous block of trapezoids to each thread (and a single thread to each core).Effectively, this partitioned the interval [a,b] into larger subintervals, and each thread simply applied the serial trapezoidal rule to its subinterval. See Figure 5.4 for an example.

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**

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. That is, only one thread can execute the following structured block at a time.

**3.5 OpenMP Directives**

1. There are enough OpenMP compiler directives and functions; to be able to parallelize a wide variety of C code segments.
2. OpenMP provides what’s known as a “directives-based” shared-memory API. In C and C++, this means that there are special preprocessor instructions known as **pragmas**. Pragmas are typically added to a system to allow behaviors that aren’t part of the basic C speciﬁcation.

This section introduces a powerful set of OpenMP compiler directives:

1. ***parallel***, which precedes a block of code to be executed in parallel by multiple threads
2. ***for***, which precedes a for loop with independent iterations that may be divided among threads executing in parallel
3. ***parallel for***, a combination of the parallel and for directives
4. ***sections***, which precedes a series of blocks that may be executed in parallel
5. ***parallel sections***, a combination of the parallel and section directives
6. ***critical***, which precedes a critical section
7. ***single***, which precedes a code block to be executed by a single thread

**1. pragma omp parallel**

1. We’ve already seen that pragmas in C and C++ start with

**# pragma**

OpenMP pragmas always begin with

**# pragma omp**

1. 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.

**# pragma omp parallel num\_threads(thread\_count)**

**2. pragma omp parallel for**

1. 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 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;

1. 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.
2. 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, that is, 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.

**3. pragma omp critical**

1. 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;**

1. 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. That is, only one thread can execute the following structured block at a time.

**The Reduction Clause**

global result = 0.0;

# pragma omp parallel num threads(thread count)

{

# pragma omp critical

global\_result += Local trap(double a, double b, int n); }

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.

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;

}

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 a 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 in order to get a single result. Furthermore, all of the intermediate results of the operation should be stored in the same variable: the reduction variable.

The syntax of the reduction clause is

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

global result = 0.0;

# pragma omp parallel num threads(thread count) \

reduction(+: global\_result)

global\_result += Local\_trap(double a, double b, int n);

**3.6 Library Functions**

**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.

**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)

**omp\_get\_thread\_num()** and **omp\_get\_num\_threads()**

Since each thread has its own stack, a thread executing the Hello function will create its own private, local variables in the function. In our example, when the function is called, each thread will get its rank or id and the number of threads in the team by calling the OpenMP functions **omp\_get\_thread\_num** and **omp\_get\_num\_threads**, respectively. The rank or id of a thread is an int that is in the range 0,1,..., thread count −1. The syntax for these functions is

**int omp\_get\_thread\_num(void);**

**int omp\_get\_num\_threads(void);**

**3.7 Handling Loops**

1. 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.
2. First, OpenMP will only parallelize for loops. It won’t parallelize while loops or do−while loops. This may not seem to be too much of a limitation, since any code that uses a while loop or a do−while loop can be converted to equivalent code that uses a for loop instead.

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;

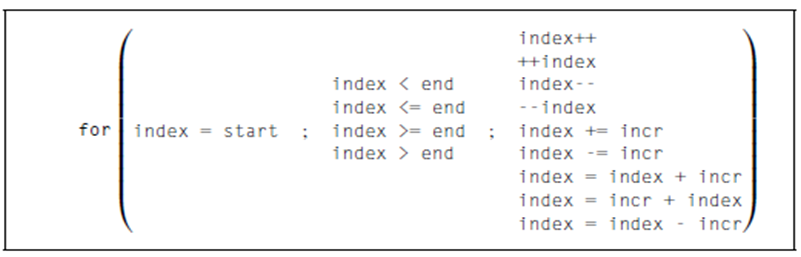
. . .

}

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.

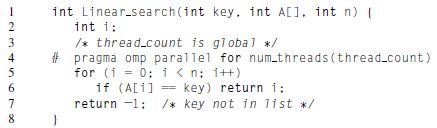
Infact,OpenMPwillonlyparallelizeforloopsthatareincanonicalform.Loops in canonical form take one of the forms shown in Program 5.3. The variables and expressions in this template are subject to some fairly obvious restrictions:

1. The variable index must have integer or pointer type (e.g., it can’t be a float).
2. The expressions start, end,and incr must have a compatible type. For example, if index is a pointer, then incr must have integer type.
3. The expressions start, end, and incr must not change during execution of the loop.
4. During execution of the loop, the variable index can only be modiﬁed by the “increment expression” in the for statement.



**Data dependences**

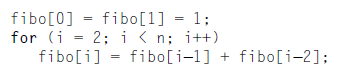
If a for loop fails to satisfy one of the rules outlined in the preceding section, the compiler will simply reject it. For example, suppose we try to compile a program with the following linear search function:



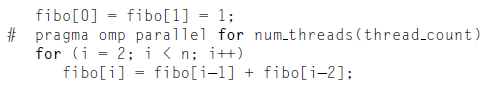
The gcc compiler reports: Line 6: error: invalid exit from OpenMP structured block

**Loop-carried dependence**

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:



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



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.

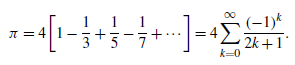
We see two important points here:

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.

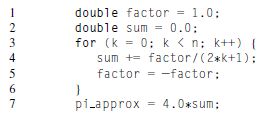
The dependence of the computation of fibo[6] on the computation of fibo[5] is called a **data dependence**. Since the value of fibo[5] is calculated in one iteration, and the result is used in a subsequent iteration, the dependence is sometimes called a **loop-carried dependence**.

**Estimating π**

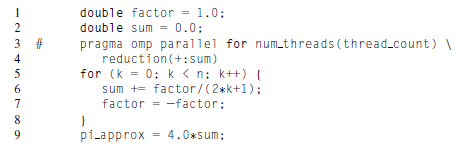
One way to get a numerical approximation to π is to use many terms in the formula



We can implement this formula in serial code with



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



we will eliminate the loop dependency. 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

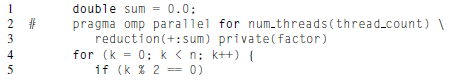
On the other hand, if we run the program with only one thread, we always get

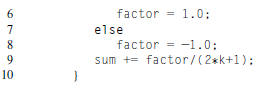
* + 1. With n = 1000 terms and 1 threads,
    2. Our estimate of pi = 3.14059265383979

**Private clause**

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

1. 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.
2. Therefore, in addition to eliminating the loop-carried dependence in the calculation of factor, we need to insure that each thread has its own copy of factor.
3. That is, in order to make our code correct, we need to also insure that factor has private scope. We can do this by adding a private clause to the parallel **for directive.**





**Accessing Private Data Outside the Parallel Region**

**firstprivate clause**

1. When a variable is declared as private, each thread gets a private copy of the variable. However, the variable does not get initialized at the start of the parallel region, and its value does not get propagated beyond the end of the region.
2. Listing 3.1 shows an example of code where the value of the variable before the parallel region is important

**Listing 3.1 Parallel Region That Accesses the Value of a Private Variable**

#include <stdio.h>

int main()

{

int data=1;

#pragma omp parallel for private(data)

for ( int i=0; i<100; i++ )

{

printf( "data=%i\n", data ); }

return 0; }

1. Although the variable data is initialized to the value one outside the parallel region, this value is not passed into the private copy that each thread obtains inside the parallel region.
2. Hence, the value that is printed is undefined. To initialize the value of the variable in the parallel region to the value it held before the region, the variable needs to be declared using the clause firstprivate(variables).
3. This tells the compiler to include code that copies the existing value into the private copy held by each thread in the parallel region. Listing 3.2 shows the modified code.

**Listing 3.2 Declaring a Variable as firstprivate to Pass Its Value into the Parallel Region**

#include <stdio.h>

int main()

{ int data=1;

#pragma omp parallel for firstprivate(data)

for ( int i=0; i<100; i++ )

{

printf( "data=%i\n", data ); }

return 0; }

**lastprivate clause**

1. Another situation is where the value of a variable is used after a parallel region. In this case, it is important to retain the value that was written into this variable by the thread that executed the last iterations of the loop. This preserves the semantics of the serial program.
2. The clause that enables this to happen is lastprivate(variables). This clause is also supported on parallel sections, which will be introduced later.
3. Listing 3.3 shows an example of using lastprivate to retain the last value written into the variable.

**Listing 3.3 Passing the Value of a Variable Out of a Parallel Region Using lastprivate**

#include <stdio.h>

int main()

{ int data=1;

#pragma omp parallel for lastprivate(data)

for ( int i=0; i<100; i++ )

{ data = i\*i;

printf( "data=%i\n", data );

}

printf( "Final value=%i\n", data );

return 0; }

**Work-sharing constructs**

**Scheduling loops**

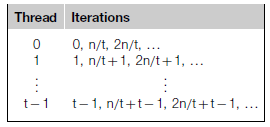
1. When we ﬁrst encountered the parallel for directive, we saw that the exact assignment of loop iterations to threads is system dependent. However, most OpenMP implementations use roughly a block partitioning: if there are n iterations in the serial loop, then in the parallel loop the ﬁrst *n/thread\_count* 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.
2. For example, suppose we want to parallelize the loop

sum = 0.0;

for (i = 0; i <= n; i++)

sum += f(i);

1. 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.
2. 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:



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 ∗/

1. The call f(i) calls the sine 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.
2. 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 was2.76seconds.This is a speedup of only1.33.However,when we ran the program with two threads and a cyclic assignment, the run-time was decreased to1.84 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.
3. In OpenMP, assigning iterations to threads is called **scheduling**, and the schedule clause can be used to assign iterations in either a parallel for or a for directive.

**The schedule clause**

1. 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:

1. **static.** The iterations can be assigned to the threads before the loop is executed.
2. **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.
3. **auto.** The compiler and/or the run-time system determine the schedule.
4. **runtime.** The schedule is determined at run-time.

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.

**The *static* schedule type**

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

Thread 0: 0,3,6,9

Thread 1: 1,4,7,10

Thread 2: 2,5,8,11

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

Thread 0: 0,1,6,7

Thread 1: 2,3,8,9

Thread 2: 4,5,10,11

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

Thread 0: 0,1,2,3

Thread 1: 4,5,6,7

Thread 2: 8,9,10,11

Thus the clause schedule(static, total\_iterations/thread\_count) ismoreor less equivalent to the default schedule used by most implementations of OpenMP. The chunksize can be omitted. If it is omitted, the chunksize is approximately total\_iterations/thread\_count.

**The *dynamic* and *guided* schedule types**

1. 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.
2. In a guided schedule, each thread also executes a chunk, and when a thread ﬁnishes a chunk, it requests another one. However, in a guided schedule, as chunks are completed, the size of the new chunks decreases.
3. 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 5.3. 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≈5000, since there are 9999 unassigned iterations. The second chunk has size 4999/2≈2500, and so on. In a guided schedule, if no chunksize is speciﬁed, the size of the chunks decreases down to 1. If chunksize is speciﬁed, it decreases down to chunksize, with the exception that the very last chunk can be smaller than chunksize.

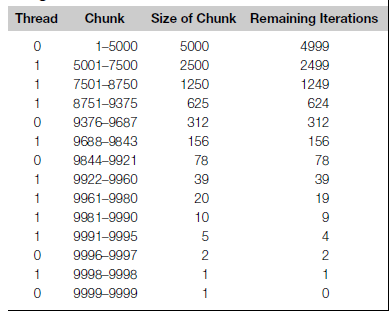


Table 5.3 Assignment of Trapezoidal Rule Iterations 1–9999 using a guided Schedule with Two Threads

double sum( double \*a, int n )

{ double total = 0.0;

#pragma omp parallel for reduction( +: total) schedule( **guided**, n/50)

for ( int i=0; i<n; i++ )

{ total += a[i];

}

return total; }

**The *runtime* schedule type**

1. 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.
2. 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.
3. For example, suppose we have a parallel for directive in a program and it has been modiﬁed by schedule(runtime). Then if we use the bash shell, we can get a cyclic assignment of iterations to threads by executing the command

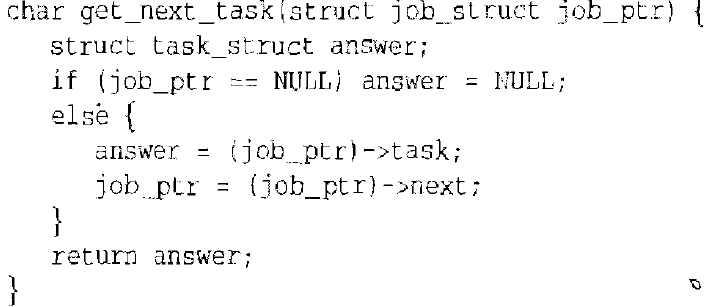
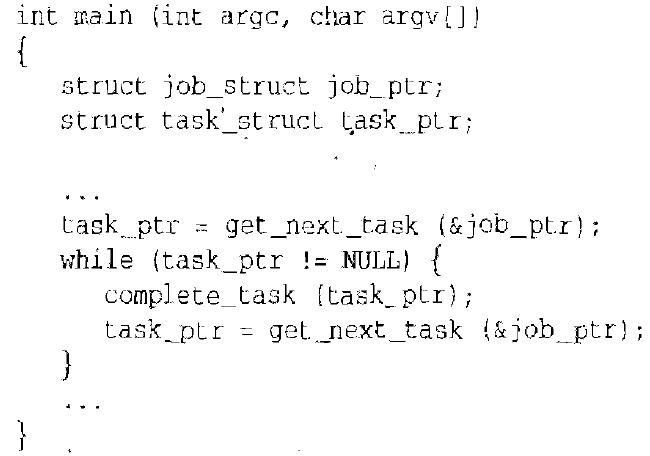
$ export OMP\_SCHEDULE="static,1"

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

**Data parallelism and functional parallelism**

**Data parallelism**

1. To this point we have focused on the parallelization of simple for loops. They are perhaps the most common opportunity for parallelism. However , we should nor ignore other opportunities for concurrency. In this section we look at two examples of data parallelism outside simple for loops.
2. First let's consider an algorithm to process a linked list of tasks.
3. 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



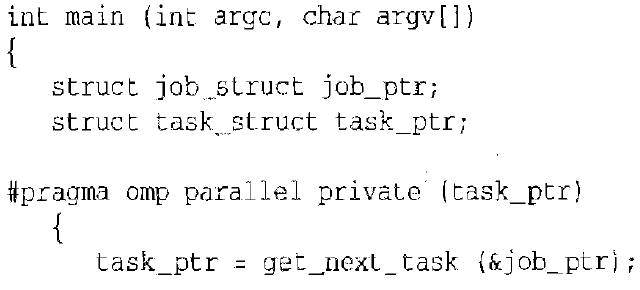
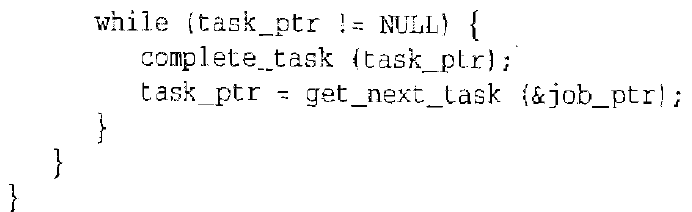
1. How would we like this algorithm to execute in parallel? We want every thread to do the same thing: repeatedly take the next task from the list and complete it, until there are 00 more tasks to do. We need to ensure that no two threads take the same task from the list. In other words, it is important to execute function get\_next\_task atomically.

***parallel* Pragma**

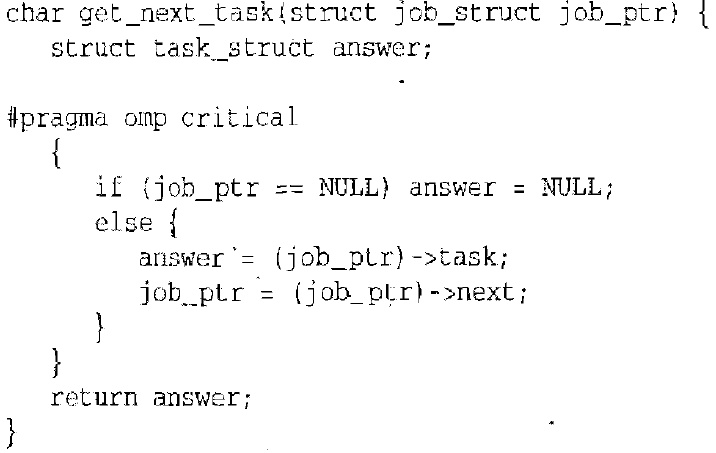
1. The parallel pragma precedes a block of code that should be executed by all of the threads. It has this syntax:

**#pragma omp parallel**

1. 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.
2. 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:



1. We use the critical pragma to ensure mutually exclusive execution of this critical section of code. Here is the rewritlen function get\_next\_task:



***for* Pragma**

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

for (i=0; i < m;i++)

{

low =c[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[j])/b[j]

}

1. 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.
2. If we put the paralle1 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**

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

#pragma omp parallel private(i, j)

for (i=0; i < m;i++)

{

low =c[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[j])/b[j]

}

***single* Pragma**

1. 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 =c[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[j])/b[j]

}

***nowait* Clause**

1. 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.
2. 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 =c[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[j])/b[j]

}

**Functional parallelism**

1. 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));

1. If all of the functions are side-effect free, we can represent the data dependences as shown in Figure 17.10. Clearly functions al pha, beta, and del t a 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.

**parallel *sections* Pragma**

1. 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**

***section* Pragma**

1. The *section* pragma precedes each block of code within the encompassing block preceded by the *parallel sections* pragrna. (The section pragma may be omitted for the first parallel section after the *parallel sections* pragma)
2. 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 10 create a block of code containing these three assignment statements.

# pragma omp parallel sections

{

#pragma omp section

v = alpha();

#pragma omp section

w = beta ();

#pragma omp section

y =delta();

}

x =gamma (v,w);

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

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

***sections* Pragma**

1. 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
2. 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 secions pragma to identify the first and second pairs of functions that may execute in parallel.
3. 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.

#pragma omp parallel

{

# pragma omp sections

{

# pragma omp section

v = alpha();

# pragma omp section

w = beta ();

}

# pragma omp sections

{

# pragma omp section

x =gamma (v,w);

# pragma omp section

y =delta();

}

}

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