Chapter 11 Shared Memory Parallel Computing Using OpenMP

Preface

Chapter 10 introduced the core concepts of shared memory parallel programming and the basics of multi-threaded programming using the Pthreads library. It is advisable to read that chapter before reading this one, because it contains important concepts about multi-threading in general.

This chapter is an introduction to *OpenMP* in C/C++. All statements about its syntax in these notes are specific to C/C++. Whereas Pthreads requires the programmer to explicitly manage user-defined threads, OpenMP removes some of the burden from the programmer, making it easier to parallelize existing sequential code, and to do so in an incremental way.

Concepts Covered

Shared memory parallelism,
threads,
tasks,
the OpenMP API

11.1 Introduction

It is hard to use a threading library to decompose a sequential program into threads that run in parallel without knowing a great deal about how to do this correctly. If some of the tasks of multithreading could be handled by the compiler, it would be make it possible to convert serial code to parallel code without needing to know all of the details of a library's API.

In the early 1990's different hardware vendors provided their own ways to add parallelism to sequential programs. A standard was needed. By 1997, a group of major computer hardware and software vendors formed the ***OpenMP Architecture Review Board*** to standardize the means by which programmers could partially convert sequential programs into multi-threaded programs with the assistance of the compiler. That year they released the first version of the ***OpenMP*** standard for FORTRAN programs, and a year later, a version for C/C++. *It was designed to make it possible for programmers to incrementally evolve a sequential program into a parallel program.* The OpenMP specification has been revised several times since its creation, with more features being added over the years. These notes describe OpenMP 4.5.

OpenMP is an open API for writing shared-memory parallel programs written in C/C++ and FORTRAN. Parallelism is achieved exclusively through the use of threads. It is portable, scalable, and supported on a wide variety of multiprocessor/core, shared memory architectures, whether they are UMA or NUMA. Because it is an open API, it has implementations on many platforms, many of which are open source, such as all of those provided by GNU under various GNU Public Licenses. It makes it possible for programmers to incrementally introduce parallelism based on threads into sequential programs.

It is important to understand what OpenMP ***does not do*** before continuing. OpenMP

- is not necessarily implemented identically by all vendors;
- is not guaranteed to make the most efficient use of shared memory;
- is not required to check for data dependencies, data conflicts, race conditions, or deadlocks;
- is not required to check for code sequences that cause a program to be classified as non-conforming;
- does not guarantee that input or output to the same file by different threads is synchronized when they run in parallel.

11.2 The Parallel Execution Model

Every OpenMP program begins with a single thread, called the ***primary***, ***initial***, or ***master*** thread, that can *fork* new threads that run in parallel and that eventually join the master thread; this is an example of the ***fork/join paradigm***, depicted in Figure 11.1. The master thread is part of the team of threads that run in parallel. OpenMP distinguishes between the threads and the work that they do. The work that a threads performs is called a ***task***; the following sections elaborate on the task concept.

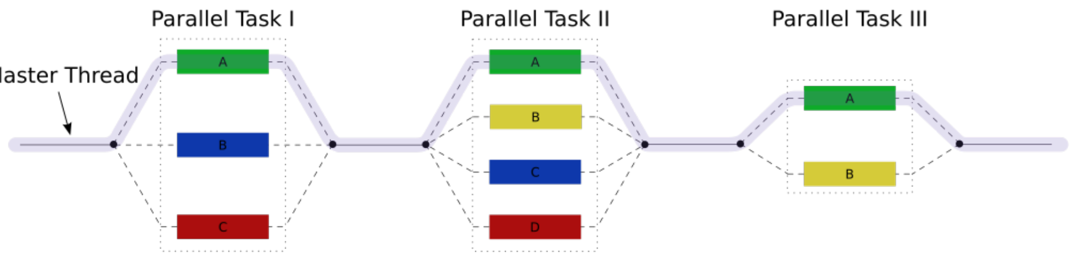

Figure 11.1: The fork-join model

The join that takes place is in effect an implicit synchronization barrier; the master thread continues executing the code after the parallel task only when all threads have completed executing their respective tasks.

11.3 The Components of the OpenMP API

The OpenMP API has three primary components:

- ***Compiler directives***. These are preprocessor directives that can be used by programmers to define and control parallel regions of code.
- ***Runtime library routines***. These are functions from the OpenMP library that can be called by the program and are then linked into the program;
- ***Environment variables***. These can be used to control the behavior of OpenMP programs.

It is possible to parallelize many sequential programs without using most of the API.

11.3.1 Basics

Pragmas

OpenMP defines a ***parallel region*** as a block of code or a single statement that may run in parallel. Programmers can insert compiler directives called ***pragmas*** into their code to identify parallel regions; these directives instruct the OpenMP run-time library to execute the region in parallel using two or more threads. Pragmas are a strict subset of compiler directives; all pragmas are directives but not vice versa (e.g. `#define` is a directive, not a pragma.) ***Pragmas*** always start with the `pragma` keyword. For example, a simple pragma is

```
#pragma omp parallel
```

which specifies that the statement or block that follows immediately after it is to be executed by some number of threads in parallel. Thus,

```
#pragma omp parallel
{
    printf(“Hello World\n”);
}
```

will cause multiple threads to execute the `printf` statement in parallel. The default number of threads will be the number of cores that the compiler detects. The set of threads that are forked to execute a given parallel region is called a ***thread team***. In Figure 11.1, there are three teams of threads: those that execute Parallel Task I, those that execute Parallel Task II, and those that execute Parallel Task III.

Library Routines

Runtime library functions look like ordinary function calls, but they are not part of the native language's library; they are part of the OpenMP runtime library. For example, `omp_get_thread_num()` is a function that returns the unique integer ID of the thread that calls it. To use these runtime routines, one needs to include the `<omp.h>` header file in the program. Runtime routines can be used to do many different tasks. Some common ones are:

- Setting and querying the number of threads
- Querying a thread's unique identifier (thread ID), a thread's ancestor's identifier, or the thread team size
- Setting and querying the dynamic threads feature (see below)
- Querying whether it is in a parallel region, and if so, at what level
- Setting and querying nested parallelism (see below)
- Setting, initializing and terminating locks and nested locks
- Querying wall clock time and resolution

Environment Variables

Environment variables are used to control how parallel code is executed. Some examples of such control include

- Setting the number of threads
- Specifying how loop iterations are divided
- Binding threads to processors
- Enabling and disabling nested parallelism; setting the maximum levels of nested parallelism
- Enabling and disabling dynamic threads
- Setting thread stack size
- Setting thread wait policy

Dynamic Threads

The API lets the runtime environment dynamically change the number of threads used to execute parallel regions for more efficient use of resources, if possible, but implementations may or may not support this feature. In implementations that support it, there is an environment variable that controls whether or not it is enabled (see `OMP_DYNAMIC`). The installed versions of *gcc* on our network all support dynamic thread creation.

Nested Parallelism

The API allows the placement of parallel regions inside other parallel regions, but implementations may or may not support this feature. In implementations that support it, there is an environment variable that controls whether or not it is enabled (see `OMP_NESTED`).

I/O

OpenMP does not specify anything about parallel I/O. An implementation may handle this as it chooses. If every thread uses a different file or device, there is no significant problem. It up to the programmer to ensure that I/O is handled correctly.

Tasks

OpenMP makes a distinction between ***threads*** and ***the work that they do***. A thread is an execution unit, like a process. It is an entity that can be scheduled by the operating system scheduler. The work that a thread performs is called its ***task***. A *task* is a specific instance of executable code and its data environment, generated when a thread encounters a construct that causes the creation of tasks, such as a `parallel` directive. You can think of a task as a chunk of work to be done. A thread may execute different tasks during its lifetime. When a thread stops executing one task and starts another, it is called ***task switching***. The converse is also true - a task may be executed by different threads before it is completed[1]. Thus, threads and tasks are really two different things.

In OpenMP, tasks may be created ***explicitly*** or ***implicitly***. The parallel directive causes the creation of ***implicit tasks***. In contrast, there is a `task` directive that can be used to create tasks explicitly, and these naturally are called ***explicit tasks***.

Compiling and Running OpenMP Programs

To use any part of the OpenMP API, one needs to tell the compiler to compile against it. With *gcc*, this is done by passing the compile-time flag `-fopenmp`. This enables the OpenMP directive `#pragma omp` in C/C++. For example:

```
gcc -fopenmp -o outputfile  my_openmp_program.c
```

is how to compile the program `my_openmp_program.c` if it has any pragmas in it[2]. If it also makes calls to the runtime library, the flag will arranges for automatic linking of the OpenMP runtime library to the program, and the program must include the header file `omp.h`. If the program does not use the library, it is not necessary to include the header file, but there is no downside to doing so other than slightly longer compile time.

A simple OpenMP/C program to demonstrate this is in Listing 11.1 below.

Listing 11.1: helloworld1_omp.c

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

int main ( int argc, char *argv[] )
{
    printf("\n  Before the parallel region.\n\n" );
    # pragma omp parallel
    {
        printf("  Hello world\n" );
    }
    printf("\n  After the parallel region.\n" );
    return 0;
}
```

If this code is in the file `helloworld1_omp.c`, and we want the executable to be stored in `helloworld1_omp`, we would compile it as follows:

```
gcc -o helloworld1_omp  helloworld1_omp.c -fopenmp
```

[1] ***Tied tasks*** are bound to the threads that start them, whereas ***untied threads*** can be assigned to a different thread.

[2] The placement of the `-f` flag in *gcc* is flexible - it can appear before or after other flags or the source code arguments.

11.3.2 Some Basic OpenMP Terminology

This is but a tiny fragment of the terms defined by the OpenMP standard. It consists of just those terms that are relevant to the subset of OpenMP described in these notes.

- A ***task*** is a specific instance of executable code and its data environment, generated when a thread encounters a `task`, `taskloop`, `parallel`, `target`, or `teams` construct or a construct that combines one or more of these. An ***explicit task*** is one created when a thread encounters a `task` or `taskloop` construct. Tasks created when a `parallel`, `target`, or `teams` construct is encountered are ***implicit tasks***.
- A ***structured block*** is an executable statement or a compound block, with a single point of entry and a single point of exit.
- A ***construct*** is an OpenMP directive and the associated structured block that it controls.
- A ***region*** is the set of all code encountered during the execution of a construct, including any called functions. In other words, if a function is called withing a region, the function's code is also part of the region. A region may also be thought of as the ***dynamic or runtime extent*** of a construct or of an OpenMP library routine. The region encountered during execution of a construct can change from one run to the next, depending on the data.
  - A ***task region*** is a region consisting of all code encountered during the execution of a task. An ***implicit*** (***explicit***) ***task region*** is the task region of an implicit (explicit) task.
  - A ***parallel region*** is a region executed simultaneously by multiple threads.
- The ***master thread*** is the thread executing the sequential part of the program and spawning the child threads.
- A ***thread team*** is a set of threads that execute a parallel region.

11.4 OpenMP Directives

Directives are instructions to the compiler. They do not appear in executable code. Their effect, however, alters the executable code. Technically speaking, when the preprocessor reads a source code file and encounters a directive, it causes code to be inserted that usually alters an action that takes place at run-time. It is too much to have to say this every time when describing what a directive does. The term "construct" refers to the executable code created by the directive and the block that follows it, so we will say things like, "when a thread reaches the construct, it does this or that." Bear this in mind when reading the rest of this section.

11.4.1 The Structure of a Directive

Every directive in OpenMP has the same form:

```
#pragma omp  <directive-name>  [optional clauses] <newline>
```

where <directive-name> is replaced by specific names such as `parallel` or `section`. The clauses that can follow a directive depend on which directive is named. *Directive names are case-sensitive!* The directive must end with a newline character. In other words, ***this is not correct:***

```
#pragma omp parallel { x = 0; }
```

because the block is on the same line as the directive. If a directive is inconvenient to type on a single line, the line can be continued by "escaping" the newline character with a preceding backslash "\" in the same way that is done in *make* files or *bash* scripts. For example:

```
#pragma omp parallel shared(num_intervals, pi_estimate) \
                     private(nthrds, id, local_pi)
```

is correct as long as after the "\" the newline is typed immediately (no space in between).

11.4.2 The `parallel` Directive

This is the most important directive to start with because with this one alone you can create parallelism in your program most easily. When a thread reaches the parallel construct, it forks multiple threads to execute the parallel region immediately following the directive. This is called ***creating a thread team***. There are several questions that should come to mind immediately. We answer them first and give examples.

When a thread reaches the directive, is it part of the team that executes the parallel region?

Yes. The thread that reaches the construct will have the thread ID 0 in the team and will be considered its master thread.

How many threads are forked?

This question does not have a simple answer. There are controls of varying precedence that determine how many threads are created. We will describe them from highest to lowest precedence, ignoring the possibilities that (1) the parallel directive has an if-clause, (2) dynamic thread creation is enabled, and that (3) the process might exceed its maximum thread capacity. We will discuss these issues after.

1. If there is a `num_threads` clause, as in

   ```
   #pragma omp parallel num_threads(8)
   ```

   then the number of threads will be the value of the argument, which can a positive integer expression. The clause requires that the expression is enclosed in parentheses. It can be any expression, such as `arraysize/10` for example.

2. Otherwise, if there is a call to the library function `omp_set_num_threads()` that has been executed prior to reaching the parallel construct, the most recent call to that function determines how many threads will be in the team. This function expects a positive integer expression. See Listing 11.2 for an example.

Listing 11.2: helloworld3_omp.c

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

int main ( int argc, char *argv[] )
{
    int default_num_threads = 6;

    if ( argc < 2 ) {
        printf("Usage: %s <num-threads>\n", argv[0]);
        exit(1);
    }

    int num_threads_requested = atoi(argv[1]);
    if ( num_threads_requested < 1 ) {
        printf("Expected a positive integer argument.\n");
        exit(1);
    }
    if ( num_threads_requested  > 12 ) {
        omp_set_num_threads(num_threads_requested /2);
    }
    else {
        omp_set_num_threads(default_num_threads);
    }

    # pragma omp parallel
    {
        printf("  Hello world\n" );
    }
    return 0;
}
```

This can be used to decide the number of threads in the team based on the command line argument.

1. If there is no `num_threads` clause and no call to `omp_set_num_threads()`, then the value of the environment variable `OMP_NUM_THREADS` determines the team size. If you add the definition

   ```
   OMP_NUM_THREADS=20,10,5
   ```

   to your environment (by modifying your `.bashrc` file for example) then top level parallel regions will use 20 threads by default, second level regions will use 10, and third level, 5.

2. Lastly, if none of the above methods exist, then the default is usually the number of cores that the compiler detects on the computer, but it is an implementation default.

What complicates this answer further is that a parallel directive, as well as other directives, can have an ***if-clause***. The if-clause can be used to enable or disable a following num_threads clause or even turn off multithreading for the following block. If an if-clause is present, it has the highest precedence. The syntax of the clause for the parallel directive is:

```
#pragma omp parallel if ( parallel: <boolean-condition> ) [other-clauses]
```

where `<boolean-condition>` is replaced by any expression that evaluates to true/false (0, non-zero in C). Other clauses may follow.

Examples:

- To turn off multithreading, put a zero for the condition:

  ```
  #pragma omp parallel if ( parallel: 0)
  ```

- To conditionally apply a `num_threads` clause if some variable named `arraysize` is greater than 200 and to turn off multithreading otherwise:

  ```
  #pragma omp parallel if ( parallel: arraysize > 200 )  num_threads(20)
  ```

Governing all of the previous discussion is that the number of threads specified by any of these methods is just a request; if the process has used too many thread resources, that request may not be honored. The details can be found in Section 2.5.1 of the OpenMP specification. The program in Listing 11.3 shows how the if-clause can be used to decide whether or not to perform an action in parallel or as a single thread based on input data. It also introduces two more library routines: `omp_get_num_threads()`, which returns the number of active threads in a parallel region, and `omp_get_thread_num()`, which returns to the calling thread its thread ID.

Listing 11.3: helloworld4_omp.c

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

int main ( int argc, char *argv[] )
{
    int default_num_threads = 6;

    if ( argc < 2 ) {
        printf("Usage: %s <pretend-array-size>\n", argv[0]);
        exit(1);
    }

    int size = atoi(argv[1]);
```

```
    if ( size < 1 ) {
        printf("Expected a positive integer argument.\n");
        exit(1);
    }

    # pragma omp parallel  if(parallel:size > 100) \
                           num_threads(default_num_threads)
    {
        int team_size = omp_get_num_threads(); /* get number of threads */
        int tid       = omp_get_thread_num();  /* get thread id */
        printf("  Hello world from thread %d out of %d threads.\n",
                  tid, team_size);
    }
    return 0;
}
```

Are there any restrictions on what statements can be in the structured block?

Yes. There cannot be any jumps into the block such as go-tos, and no jumps out of the block. It must be a single-entry/single-exit block. In addition, it cannot span multiple functions or source code files.

11.4.2.1 Sharing Attributes of Variables in `parallel` Directives

All variables that are declared outside of a parallel construct are ***shared*** by default. For example, in the following code

```
int x = 0;
#pragma omp parallel
{
    x = x + 1;
}
```

`x` is shared by all threads that execute the parallel region. This code could cause a race condition of course.

The parallel directive has clauses that let you control the data-sharing attributes of variables, i.e., which variables are shared, which are private to each thread, and which are something else, as you will soon see. Each of the following clauses[3] requires a comma-separated list of variables as its argument. Each can be repeated any number of times.

- `private(`*`list`*`)` declares that the variables in the list are private to each thread in the team. Each has its own ***uninitialized*** copy of that variable.

[3]There are other clauses not explained here.

- `firstprivate(`*`list`*`)` declares that the variables in the list are private to each thread, and ***initializes*** each of them with the value that the corresponding original variable has when the construct is encountered.
- `shared(`*`list`*`)` declares that the variables in the list are shared by all threads in the team. There is a single copy of the variable.
- `reduction(reduction-identifier:` *`list`*`)` specifies a reduction operation to be performed on one or more list items. For each list item, a private copy is created in each thread, which is initialized with the initializer value of the given reduction-identifier. (For example, 0 is the initializer for summation whereas 1 is the initializer for multiplication. In short the identity element of the operation.) After the end of the region, the original list item is updated with the values of the private copies using the combiner associated with the reduction-identifier. An example will follow below.

There are many restrictions that apply to these clauses; these notes are not intended to replace reading the specification for all of the nuances regarding their use, such as whether you can list an element of an array or a member of a struct, or a const-qualified variable, etc.

Examples

An example to illustrate the use of the `firstprivate` clause and the `reduction` clause follows.

Listing 11.4: reduction_demo.c

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

int main( int argc, char *argv[] )
{
    int x   = 2;
    int sum = 0;
    int numthreads;

    if ( argc < 2 ) {
        printf("Usage: %s number-of-threads\n", argv[0]);
        exit(1);
    }

    if ( 0 >= (numthreads = atoi(argv[1]) ) ) {
        fprintf(stderr,"Number of threads is not a positive integer.\n");
        exit(1);
    }

    omp_set_num_threads(numthreads);
    #pragma omp parallel firstprivate(x) reduction(+:sum)
    {
        x   = x * omp_get_thread_num();
        if ( omp_get_thread_num() == 0)
            numthreads = omp_get_num_threads();
            sum += x;
            }
    printf("Double the sum of the thread ids of %d threads is %d\n",
                numthreads, sum);
    return 0;
}
```

Notes

- `firstprivate(x)` initializes the copy of `x` in each spawned thread to its value when the region is reached, which is 2.
- `reduction(+:sum)` causes each thread to have its own copy of `sum` and performs an addition operation, in this case

  ```
  sum = sum + x
  ```

  in each thread. The effect is that each threads add the initial value of `x` to `sum`, initially 0, so that its final value is `numthreads*x`.

11.4.3 Worksharing Constructs

Imagine for a moment that you have a parallel region of code such as the following:

```
#pragma omp parallel
{
   /*
      stuff to be done in parallel
   */
}
```

Within such a region, you can specify what work the various threads will perform in a very explicit way. A ***worksharing construct*** is a type of directive that distributes the execution of its associated region among the members of the thread team that encounters it. Each thread in the team executes its portion of the region in the context of the implicit task that it was executing when it encounters the worksharing construct. This means, for example, that it still has access to the same variables it had before reaching that construct.

There is no implicit barrier at the beginning of a worksharing construct. Threads that encounter it before others can enter it immediately. On the other hand, *there is an implicit barrier at the end of the construct*; no thread advances past it until all threads complete their tasks.

There are four worksharing constructs:

- *loop* construct
- `sections` construct
- `single` construct

- `workshare` construct

The work units in these worksharing constructs are tightly controlled, either by an iteration limit and limited scheduling, or by a limited number of sections or single regions. Worksharing constructs were designed mostly for highly data parallel computing. This is in contrast to task constructs, which were created to support task parallelism.

The most useful worksharing construct is the ***loop construct***. It is simple to use and gives you the ability to parallelize much of your sequential program easily. We begin with it.

11.4.3.1 Loop Construct

The purpose of the loop construct is to allow the separate iterations of a for-loop to be executed in parallel. The `for` directive is the one that accomplishes this. It specifies that the iterations of the loops that follow it will be executed in parallel by the threads in the team executing the parallel region containing the `for` directive[4] It is intended to be nested inside a construct that creates multiple threads, i.e., the `parallel` directive. Its syntax is

```
#pragma omp for [optional clauses]  <newline>
    for-loops
```

The most important requirement concerns the structure of the enclosed for-loops. The loops must be in what OpenMP calls canonical form. Essentially, the loops must be in a form that makes it possible for the compiler to parallelize the loop easily. It allows the iteration count of all associated loops to be computed before executing the outermost containing loop. The canonical form of a loop is

```
for (init-expr; test-expr; incr-expr) structured-block
```

where

- *`init-expr`* is one of

  ```
  var = lb
  integer-type var = lb
  random-access-iterator-type var = lb
  pointer-type var = lb
  ```

  in which *`lb`* is an expression whose value is loop-invariant (it does not change as the loop iterates.)

- *`test-expr`* is either

  ```
  var relational-operator b
  b relational-operator var
  ```

[4]Technically, the threads are those in the parallel region to which the for binds according to the nesting rules, which will be the innermost enclosing parallel region.

in which *b* is an expression whose value is loop-invariant and the only relational operators allowed are <, <=, >, and >=.

- *incr-expr* is one of :

```
++var
var++
- - var
var - -
var += incr
var - = incr
var = var + incr
var = incr + var
var = var - incr
```

  in which *incr* is an expression whose value is loop-invariant and var is either an integer-type or a pointer-type variable or in C++, a random-access iterator.

The other restrictions are fairly intuitive - because the compiler has to compute the number of iterations, the test expression must have the right sense with respect to the increment. For example, this does not work:

```
for ( i = M; i > N; i++ )
```

because the variable is increasing and the test compares `i` with a "greater-than" check against `N`; the loop may never end.

The loop variable becomes private automatically in this construct; each thread has its own private copy of it. Also, it must not be modified during the execution of the for-loop other than in *incr-expr*. Its value after the loop is unspecified unless a `lastprivate` or `linear` clause is specified.

Some of the possible clauses are

- `private(`*list*`)` has the same meaning as described in Section 11.4.2.
- `firstprivate(`*list*`)` has the same meaning as described in Section 11.4.2.
- `lastprivate(`*list*`)` declares that the variables in the list are private to each thread, and causes the corresponding original list item to be updated after the end of the region.
- `reduction(reduction-identifier: `*list*`)` has the same meaning as described in Section 11.4.2.
- `schedule([`*modifier* `[, `*modifier*`]:]`*kind*`[, `*chunk_size*`])` specifies how iterations of these associated loops are divided into contiguous non-empty subsets, called ***chunks***, and how these chunks are distributed among threads of the team.
- `nowait`: If a `nowait` clause is present, an implementation may omit the barrier at the end of the worksharing region. In this case, threads that finish early may proceed straight to the instructions following the worksharing region without waiting for the other members of the team to finish the worksharing region.

Examples will now illustrate the loop construct.

Listing 11.5: parallel_for_demo1.c

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

int main( int argc, char* argv[] )
{
    int *nums;
    int array_size;
    int i;

    if ( argc < 2 ) {
        printf("usage: %s array-size\n",argv[0]);
        exit(1);
    }
    array_size = atoi(argv[1]);
    if ( 0 >= array_size )
        printf("array size must be a positive integer\n");
        exit(1);
    }
    nums = malloc(array_size*sizeof(int));
    if ( NULL == nums ) {
        fprintf(stderr,"Error allocating memory for array\n");
        exit(1);
    }

    #pragma omp parallel shared(array_size,nums)
       #pragma omp for
       for (i = 0; i< array_size; i++)
           /* array is filled with id of thread that filled it */
           nums[i] = omp_get_thread_num();

    /* master thread prints out the array */
    for ( i = 0; i < array_size; i++ )
        printf("%d\n",nums[i]);

    free(nums);
    return 0;
}
```

In Listing 11.5, the for-loop iterations are assigned to the default number of threads. Each thread writes its thread ID into the array entry whose index is the loop iteration. The master thread prints out the array contents so that we can see which threads were assigned which tasks (loop iterations.) You will see when you run this that the number is usually divided evenly among the threads.

The `schedule` clause allows us to change which threads execute which chunks. The preceding program is modified by inclusion of two different `schedule` clauses in the next listing. One requests a static schedule but changes the chunk size and the next requests a dynamic schedule. The program prints out array contents that show the different schedules.

Listing 11.6: parallel_for_demo2.c

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

int main( int argc, char* argv[] )
{
    int *nums;
    int *nums2;
    int array_size;
    int i;

    if ( argc < 2 ) {
        printf("usage: %s array-size\n",argv[0]);
        exit(1);
    }

    array_size = atoi(argv[1]);
    if ( 0 >= array_size ) {
        printf("array size must be a positive integer\n");
        exit(1);
    }
    nums  = malloc(array_size*sizeof(int));
    nums2 = malloc(array_size*sizeof(int));
    if ( NULL == nums || NULL == nums2 ) {
        fprintf(stderr,"Error allocating memory for array\n");
        exit(1);
    }

    /* declare a parallel region */
    #pragma omp parallel for shared(array_size,nums) schedule(static,2)
       for (i = 0; i< array_size; i++)
           /* array is filled with id of thread that filled it */
           nums[i] = omp_get_thread_num();


    #pragma omp parallel for shared(array_size,nums) schedule(dynamic,2)
       for (i = 0; i< array_size; i++)
           /* array is filled with id of thread that filled it */
           nums2[i] = omp_get_thread_num();

    /* master thread prints out the two arrays showing the difference
       between static and dynamic schedules */
    printf("Static\tDynamic\n");
    for ( i = 0; i < array_size; i++ )
```

```
        printf("%d\t%d\n",nums[i], nums2[i]);

    free(nums);
    free(nums2);
    return 0;
}
```

The last example is designed to show that there is no implicit barrier at the start of the loop construct and that the threads do not necessarily get assigned chunks that are related to their thread IDs. It delays the different threads by different amounts and within the loop

Listing 11.7: parallel_for_demo3.c

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

int main( int argc, char* argv[] )
{
    int *nums;
    int array_size;
    int i;
    int tid;

    if ( argc < 2 ) {
        printf("usage: %s array-size\n",argv[0]);
        exit(1);
    }
    array_size = atoi(argv[1]);
    if ( 0 >= array_size ) {
        printf("array size must be a positive integer\n");
        exit(1);
    }
    nums = calloc(array_size,sizeof(int));
    if ( NULL == nums ) {
        fprintf(stderr,"Error allocating memory for array\n");
        exit(1);
    }

    /* declare a parallel region */
    #pragma omp parallel shared(array_size,nums) private(tid) num_threads(8)
    {
       tid = omp_get_thread_num();
       usleep(tid*800000);
    /* declare a for to cause threads to execute loop iterations */
       #pragma omp for
       for (i = 0; i< array_size; i++) {
           usleep((omp_get_num_threads()-tid)*800000);
```

```
                printf("Thread %d in loop iteration %d\n",tid,i);
                /* array is filled with id of thread that filled it */
                nums[i] = tid;
            }
    }

    /* master thread prints out the array */
    for ( i = 0; i < array_size; i++ )
        printf("%d\n",nums[i]);

    free(nums);
    return 0;
}
```