7.10 OpenMP: a Standard for Directive Based Parallel Programming

In the first part of this chapter, we studied the use of threaded APIs for programming shared address space machines. While standardization and support for these APIs has come a long way, their use is still predominantly restricted to system programmers as opposed to application programmers. One of the reasons for this is that APIs such as Pthreads are considered to be low-level primitives. Conventional wisdom indicates that a large class of applications can be efficiently supported by higher level constructs (or directives) which rid the programmer of the mechanics of manipulating threads. Such directive-based languages have existed for a long time, but only recently have standardization efforts succeeded in the form of OpenMP. OpenMP is an API that can be used with FORTRAN, C, and C++ for programming shared address space machines. OpenMP directives provide support for concurrency, synchronization, and data handling while obviating the need for explicitly setting up mutexes, condition variables, data scope, and initialization. We use the OpenMP C API in the rest of this chapter.

7.10.1 The OpenMP Programming Model

We initiate the OpenMP programming model with the aid of a simple program. OpenMP directives in C and C++ are based on the #pragma compiler directives. The directive itself consists of a directive name followed by clauses.

```
#pragma omp directive [clause list]
```

OpenMP programs execute serially until they encounter the parallel directive. This directive is responsible for creating a group of threads. The exact number of threads can be specified in the directive, set using an environment variable, or at runtime using OpenMP functions. The main thread that encounters the parallel directive becomes the ***master*** of this group of threads and is assigned the thread id 0 within the group. The parallel directive has the following prototype:

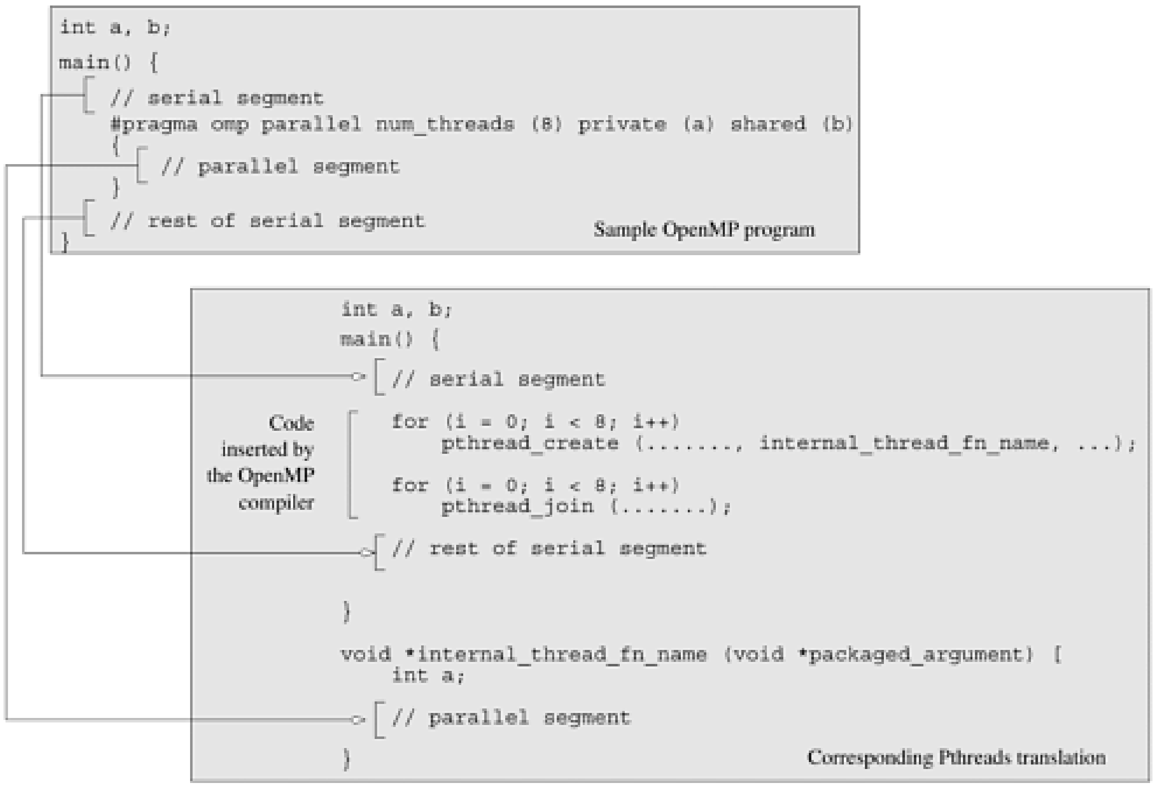
```
#pragma omp parallel [clause list]
/* structured block */

```

Each thread created by this directive executes the structured block specified by the parallel directive. The clause list is used to specify conditional parallelization, number of threads, and data handling.

- **Conditional Parallelization:** The clause if (scalar expression) determines whether the parallel construct results in creation of threads. Only one if clause can be used with a parallel directive.
- **Degree of Concurrency:** The clause num_threads (integer expression) specifies the number of threads that are created by the parallel directive.
- **Data Handling:** The clause private (variable list) indicates that the set of variables specified is local to each thread – i.e., each thread has its own copy of each variable in the list. The clause firstprivate (variable list) is similar to the private clause, except the values of variables on entering the threads are initialized to corresponding values before the parallel directive. The clause shared (variable list) indicates that all variables in the list are shared across all the threads, i.e., there is only one copy. Special care must be taken while handling these variables by threads to ensure serializability.

It is easy to understand the concurrency model of OpenMP when viewed in the context of the corresponding Pthreads translation. In Figure 7.4, we show one possible translation of an OpenMP program to a Pthreads program. The interested reader may note that such a translation can easily be automated through a Yacc or CUP script.

Figure 7.4. A sample OpenMP program along with its Pthreads translation that might be performed by an OpenMP compiler.


Sample OpenMP program
Code inserted by the OpenMP compiler
Corresponding Pthreads translation

```
int a, b;
main() {
    // serial segment
    #pragma omp parallel num_threads (8) private (a) shared (b)
    {
        // parallel segment
    }
    // rest of serial segment
}
```

```
int a, b;
main() {
    // serial segment
    for (i = 0; i < 8; i++)
        pthread_create (......., internal_thread_fn_name, ...);

    for (i = 0; i < 8; i++)
        pthread_join (.......);
    // rest of serial segment

}
void *internal_thread_fn_name (void *packaged_argument) [
    int a;
    // parallel segment
}
```

Example 7.9 Using the parallel directive

```
#pragma omp parallel if (is_parallel == 1) num_threads(8) \
                 private (a) shared (b) firstprivate(c)
{
    /* structured block */
}
```

Here, if the value of the variable is_parallel equals one, eight threads are created. Each of these threads gets private copies of variables a and c, and shares a single value of variable b. Furthermore, the value of each copy of c is initialized to the value of c before the parallel directive.

The default state of a variable is specified by the clause default (shared) or default (none). The clause default (shared) implies that, by default, a variable is shared by all the threads. The clause default (none) implies that the state of each variable used in a thread must be explicitly specified. This is generally recommended, to guard against errors arising from unintentional concurrent access to shared data.

Just as firstprivate specifies how multiple local copies of a variable are initialized inside a thread, the reduction clause specifies how multiple local copies of a variable at different threads are combined into a single copy at the master when threads exit. The usage of the reduction clause is reduction (operator: variable list). This clause performs a reduction on the scalar variables specified in the list using the operator. The variables in the list are implicitly specified as being private to threads. The operator can be one of +, *, -, &, |, ^, &&, and ||.

Example 7.10 Using the reduction clause

```
#pragma omp parallel reduction(+: sum) num_threads(8)
{
    /* compute local sums here */
}
/* sum here contains sum of all local instances of sums */
```

In this example, each of the eight threads gets a copy of the variable sum. When the threads exit, the sum of all of these local copies is stored in the single copy of the variable (at the master thread). ■

In addition to these data handling clauses, there is one other clause, copyin. We will describe this clause in Section 7.10.4 after we discuss data scope in greater detail.

We can now use the parallel directive along with the clauses to write our first OpenMP program. We introduce two functions to facilitate this. The omp_get_num_threads() function returns the number of threads in the parallel region and the omp_get_thread_num() function returns the integer i.d. of each thread (recall that the master thread has an i.d. 0).

Example 7.11 Computing PI using OpenMP directives

Our first OpenMP example follows from Example 7.2, which presented a Pthreads program for the same problem. The parallel directive specifies that all variables except npoints, the total number of random points in two dimensions across all threads, are local. Furthermore, the directive specifies that there are eight threads, and the value of sum after all threads complete execution is the sum of local values at each thread. The function omp_get_num_threads is used to determine the total number of threads. As in Example 7.2, a for loop generates the required number of random points (in two dimensions) and determines how many of them are within the prescribed circle of unit diameter.

```
/* ******************************************************
   An OpenMP version of a threaded program to compute PI.
   ****************************************************** */

    #pragma omp parallel default(private) shared (npoints) \
                         reduction(+: sum) num_threads(8)
    {
      num_threads = omp_get_num_threads();
      sample_points_per_thread = npoints / num_threads;
      sum = 0;
      for (i = 0; i < sample_points_per_thread; i++) {
        rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
        rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
            (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
            sum ++;
      }
    }
```

■

Note that this program is much easier to write in terms of specifying creation and termination of threads compared to the corresponding POSIX threaded program.

7.10.2 Specifying Concurrent Tasks in OpenMP

The parallel directive can be used in conjunction with other directives to specify concurrency across iterations and tasks. OpenMP provides two directives – for and sections – to specify concurrent iterations and tasks.

The for Directive

The for directive is used to split parallel iteration spaces across threads. The general form of a for directive is as follows:

```
#pragma omp for [clause list]
    /* for loop */

```

The clauses that can be used in this context are: private, firstprivate, lastprivate, reduction, schedule, nowait, and ordered. The first four clauses deal with data handling and have identical semantics as in the case of the parallel directive. The lastprivate clause deals with how multiple local copies of a variable are written back into a single copy at the end of the parallel for loop. When using a for loop (or sections directive as we shall see) for farming work to threads, it is sometimes desired that the last iteration (as defined by serial execution) of the for loop update the value of a variable. This is accomplished using the lastprivate directive.

Example 7.12 Using the for directive for computing p

Recall from Example 7.11 that each iteration of the for loop is independent, and can be executed concurrently. In such situations, we can simplify the program using the for directive. The modified code segment is as follows:

```
#pragma omp parallel default(private) shared (npoints) \
                    reduction(+: sum) num_threads(8)
{
  sum=0;
  #pragma omp for
  for (i = 0; i < npoints; i++) {
    rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
    rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
    if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
        (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
        sum ++;
  }
 }
```

The for directive in this example specifies that the for loop immediately following the directive must be executed in parallel, i.e., split across various threads. Notice that the loop index goes from 0 to npoints in this case, as opposed to sample_points_per_thread in Example 7.11. The loop index for the for directive is assumed to be private, by default. It is interesting to note that the only difference between this OpenMP segment and the corresponding serial code is the two directives. This example illustrates how simple it is to convert many serial programs into OpenMP-based threaded programs. ■

Assigning Iterations to Threads

The schedule clause of the for directive deals with the assignment of iterations to threads. The general form of the schedule directive is schedule(scheduling_class[, parameter]). OpenMP supports four scheduling classes: static, dynamic, guided, and runtime.

Example 7.13 Scheduling classes in OpenMP – matrix multiplication.

We explore various scheduling classes in the context of dense matrix multiplication. The code for multiplying two matrices a and b to yield matrix c is as follows:

```
for (i = 0; i < dim; i++) {
    for (j = 0; j < dim; j++) {
        c(i,j) = 0;
        for (k = 0; k < dim; k++) {
            c(i,j) += a(i, k) * b(k, j);
        }
    }
}
```

The code segment above specifies a three-dimensional iteration space providing us with an ideal example for studying various scheduling classes in OpenMP. ■

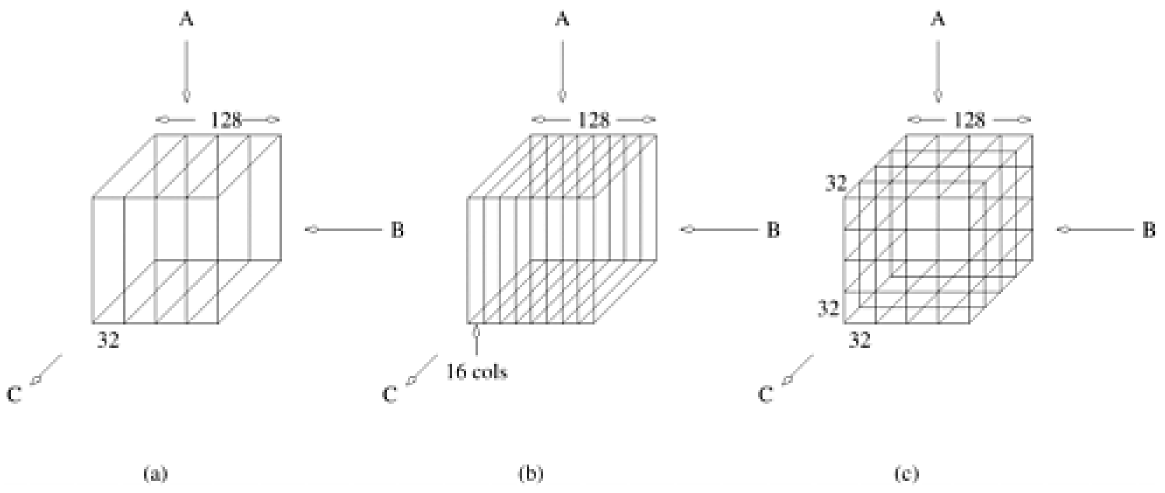
Static The general form of the static scheduling class is schedule(static[, chunk-size]). This technique splits the iteration space into equal chunks of size chunk-size and assigns them to threads in a round-robin fashion. When nochunk-size is specified, the iteration space is split into as many chunks as there are threads and one chunk is assigned to each thread.

Example 7.14 Static scheduling of loops in matrix multiplication

The following modification of the matrix-multiplication program causes the outermost iteration to be split statically across threads as illustrated in Figure 7.5(a).

```
#pragma omp parallel default(private) shared (a, b, c, dim) \
                     num_threads(4)
    #pragma omp for schedule(static)
    for (i = 0; i < dim; i++) {
        for (j = 0; j < dim; j++) {
            c(i,j) = 0;
            for (k = 0; k < dim; k++) {
                c(i,j) += a(i, k) * b(k, j);
            }
        }
    }
```

Figure 7.5. Three different schedules using the static scheduling class of OpenMP.



Since there are four threads in all, if dim = 128, the size of each partition is 32 columns, since we have not specified the chunk size. Using schedule(static, 16) results in the partitioning of the iteration space illustrated in Figure 7.5(b). Another example of the split illustrated in Figure 7.5(c) results when each for loop in the program in Example 7.13 is parallelized across threads with a schedule(static) and nested parallelism is enabled (see Section 7.10.6).

Dynamic Often, because of a number of reasons, ranging from heterogeneous computing resources to non-uniform processor loads, equally partitioned workloads take widely varying execution times. For this reason, OpenMP has a dynamic scheduling class. The general form of this class is schedule(dynamic[, chunk-size]). The iteration space is partitioned into chunks given by chunk-size. However, these are assigned to threads as they become idle. This takes care of the temporal imbalances resulting from static scheduling. If no chunk-size is specified, it defaults to a single iteration per chunk.

Guided Consider the partitioning of an iteration space of 100 iterations with a chunk size of 5. This corresponds to 20 chunks. If there are 16 threads, in the best case, 12 threads get one chunk each and the remaining four threads get two chunks. Consequently, if there are as many processors as threads, this assignment results in considerable idling. The solution to this problem (also referred to as an ***edge effect***) is to reduce the chunk size as we proceed through the computation. This is the principle of the guided scheduling class. The general form of this class is schedule(guided[, chunk-size]).In this class, the chunk size is reduced exponentially as each chunk is dispatched to a thread. The chunk-size refers to the smallest chunk that should be dispatched. Therefore, when the number of iterations left is less than chunk-size, the entire set of iterations is dispatched at once. The value of chunk-size defaults to one if none is specified.

Runtime Often it is desirable to delay scheduling decisions until runtime. For example, if one would like to see the impact of various scheduling strategies to select the best one, the scheduling can be set to runtime. In this case the environment variable OMP_SCHEDULE determines the scheduling class and the chunk size.

When no scheduling class is specified with the omp for directive, the actual scheduling technique is not specified and is implementation dependent. The for directive places additional restrictions on the for loop that follows. For example, it must not have a break statement, the loop control variable must be an integer, the initialization expression of the for loop must be an integer assignment, the logical expression must be one of <, $\leq$, >, or $\geq$, and the increment expression must have integer increments or decrements only. For more details on these restrictions, we refer the reader to the OpenMP manuals.

Synchronization Across Multiple for Directives

Often, it is desirable to have a sequence of for-directives within a parallel construct that do not execute an implicit barrier at the end of each for directive. OpenMP provides a clause – nowait, which can be used with a for directive to indicate that the threads can proceed to the next statement without waiting for all other threads to complete the for loop execution. This is illustrated in the following example:

Example 7.15 Using the nowait clause

Consider the following example in which variable name needs to be looked up in two lists – current_list and past_list. If the name exists in a list, it must be processed accordingly. The name might exist in both lists. In this case, there is no need to wait for all threads to complete execution of the first loop before proceeding to the second loop. Consequently, we can use the nowait clause to save idling and synchronization overheads as follows:

```
#pragma omp parallel
{
    #pragma omp for nowait
        for (i = 0; i < nmax; i++)
            if (isEqual(name, current_list[i])
                processCurrentName(name);
    #pragma omp for
        for (i = 0; i < mmax; i++)
            if (isEqual(name, past_list[i])
            processPastName(name);
    }
```

The sections Directive

The for directive is suited to partitioning iteration spaces across threads. Consider now a scenario in which there are three tasks (taskA, taskB, and taskC) that need to be executed. Assume that these tasks are independent of each other and therefore can be assigned to different threads. OpenMP supports such non-iterative parallel task assignment using the sections directive. The general form of the sections directive is as follows:

```
#pragma omp sections [clause list]
{
    [#pragma omp section
        /* structured block */
    ]
    [#pragma omp section
        /* structured block */
    ]
    ...
}
```

This sections directive assigns the structured block corresponding to each section to one thread (indeed more than one section can be assigned to a single thread). The clause list may include the following clauses – private, firstprivate, lastprivate, reduction, and no wait. The syntax and semantics of these clauses are identical to those in the case of the for directive. The lastprivate clause, in this case, specifies that the last section (lexically) of the sections directive updates the value of the variable. The nowait clause specifies that there is no implicit synchronization among all threads at the end of the sections directive.

For executing the three concurrent tasks taskA, taskB, and taskC, the corresponding sections directive is as follows:

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            taskA();
        }
        #pragma omp section
        {
            taskB();
        }
        #pragma omp section
        {
            taskC();
        }
    }
}
```

If there are three threads, each section (in this case, the associated task) is assigned to one thread. At the end of execution of the assigned section, the threads synchronize (unless the nowait clause is used). Note that it is illegal to branch in and out of section blocks.

Merging Directives

In our discussion thus far, we use the directive parallel to create concurrent threads, and for and sections to farm out work to threads. If there was no parallel directive specified, the for and sections directives would execute serially (all work is farmed to a single thread, the master thread). Consequently, for and sections directives are generally preceded by the parallel directive. OpenMP allows the programmer to merge the parallel directives to parallel for and parallel sections, re-spectively. The clause list for the merged directive can be from the clause lists of either the parallel or for / sections directives.

For example:

```
#pragma omp parallel default (private) shared (n)
{
    #pragma omp for
    for (i = 0 < i < n; i++) {
    /* body of parallel for loop */
    }
}
```

is identical to:

```
#pragma omp parallel for default (private) shared (n)
{
    for (i = 0 < i < n; i++) {
    /* body of parallel for loop */
    }
}

```

and:

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            taskA();
        }
        #pragma omp section
         {
            taskB();
         }
         /* other sections here */
     }
 }
```

is identical to:

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        taskA();
    }
    #pragma omp section
    {
        taskB();
     }
     /* other sections here */
 }
```

Nesting parallel Directives

Let us revisit Program 7.13. To split each of the for loops across various threads, we would modify the program as follows:

```
#pragma omp parallel for default(private) shared (a, b, c, dim) \
                     num_threads(2)
  for (i = 0; i < dim; i++) {
  #pragma omp parallel for default(private) shared (a, b, c, dim) \
                     num_threads(2)
      for (j = 0; j < dim; j++) {
          c(i,j) = 0;
          #pragma omp parallel for default(private) \
                     shared (a, b, c, dim) num_threads(2)
           for (k = 0; k < dim; k++) {
               c(i,j) += a(i, k) * b(k, j);
           }
       }
   }
```

We start by making a few observations about how this segment is written. Instead of nesting three for directives inside a single parallel directive, we have used three parallel for directives. This is because OpenMP does not allow for, sections, and single directives that bind to the same parallel directive to be nested. Furthermore, the code as written only generates a logical team of threads on encountering a nested parallel directive. The newly generated logical team is still executed by the same thread corresponding to the outer parallel directive. To generate a new set of threads, nested parallelism must be enabled using the OMP_NESTED environment variable. If the OMP_NESTED environment variable is set to FALSE, then the inner parallel region is serialized and executed by a single thread. If the OMP_NESTED environment variable is set to TRUE, nested parallelism is enabled. The default state of this environment variable is FALSE, i.e., nested parallelism is disabled. OpenMP environment variables are discussed in greater detail in Section 7.10.6.

There are a number of other restrictions associated with the use of synchronization constructs in nested parallelism. We refer the reader to the OpenMP manual for a discussion of these restrictions.

7.10.3 Synchronization Constructs in OpenMP

In Section 7.5, we described the need for coordinating the execution of multiple threads. This may be the result of a desired execution order, the atomicity of a set of instructions, or the need for serial execution of code segments. The Pthreads API supports mutexes and condition variables. Using these we implemented a range of higher level functionality in the form of read-write locks, barriers, monitors, etc. The OpenMP standard provides this high-level functionality in an easy-to-use API. In this section, we will explore these directives and their use.

Synchronization Point: The barrier Directive

A barrier is one of the most frequently used synchronization primitives. OpenMP provides a barrier directive, whose syntax is as follows:

```
#pragma omp barrier
```

On encountering this directive, all threads in a team wait until others have caught up, and then release. When used with nested parallel directives, the barrier directive binds to the closest parallel directive. For executing barriers conditionally, it is important to note that a barrier directive must be enclosed in a compound statement that is conditionally executed. This is because pragmas are compiler directives and not a part of the language. Barriers can also be effected by ending and restarting parallel regions. However, there is usually a higher overhead associated with this. Consequently, it is not the method of choice for implementing barriers.

Single Thread Executions: The single and master Directives

Often, a computation within a parallel section needs to be performed by just one thread. A simple example of this is the computation of the mean of a list of numbers. Each thread can compute a local sum of partial lists, add these local sums to a shared global sum, and have one thread compute the mean by dividing this global sum by the number of entries in the list. The last step can be accomplished using a single directive.

A single directive specifies a structured block that is executed by a single (arbitrary) thread. The syntax of the single directive is as follows:

```
#pragma omp single [clause list]
         structured block
```

The clause list can take clauses private, firstprivate, and nowait. These clauses have the same semantics as before. On encountering the single block, the first thread enters the block. All the other threads proceed to the end of the block. If the nowait clause has been specified at the end of the block, then the other threads proceed; otherwise they wait at the end of the single block for the thread to finish executing the block. This directive is useful for computing global data as well as performing I/O.

The master directive is a specialization of the single directive in which only the master thread executes the structured block. The syntax of the master directive is as follows:

```
#pragma omp master
         structured block
```

In contrast to the single directive, there is no implicit barrier associated with the master directive.

Critical Sections: The critical and atomic Directives

In our discussion of Pthreads, we had examined the use of locks to protect critical regions – regions that must be executed serially, one thread at a time. In addition to explicit lock management (Section 7.10.5), OpenMP provides a critical directive for implementing critical regions. The syntax of a critical directive is:

```
#pragma omp critical [(name)]
         structured block
```

Here, the optional identifier name can be used to identify a critical region. The use of name allows different threads to execute different code while being protected from each other.

Example 7.16 Using the critical directive for producer-consumer threads

Consider a producer-consumer scenario in which a producer thread generates a task and inserts it into a task-queue. The consumer thread extracts tasks from the queue and executes them one at a time. Since there is concurrent access to the task-queue, these accesses must be serialized using critical blocks. Specifically, the tasks of inserting and extracting from the task-queue must be serialized. This can be implemented as follows:

```
#pragma omp parallel sections
{
   #pragma parallel section
   {
      /* producer thread */
      task = produce_task();
      #pragma omp critical ( task_queue)
      {
         insert_into_queue(task);
```

```
        }
    }
    #pragma parallel section
    {
        /* consumer thread */
        #pragma omp critical ( task_queue)
        {
            task = extract_from_queue(task);
        }
        consume_task(task);
    }
}
```

Note that queue full and queue empty conditions must be explicitly handled here in functions insert_into_queue and extract_from_queue. ■

The critical directive ensures that at any point in the execution of the program, only one thread is within a critical section specified by a certain name. If a thread is already inside a critical section (with a name), all others must wait until it is done before entering the named critical section. The name field is optional. If no name is specified, the critical section maps to a default name that is the same for all unnamed critical sections. The names of critical sections are global across the program.

It is easy to see that the critical directive is a direct application of the corresponding mutex function in Pthreads. The name field maps to the name of the mutex on which the lock is performed. As is the case with Pthreads, it is important to remember that critical sections represent serialization points in the program and therefore we must reduce the size of the critical sections as much as possible (in terms of execution time) to get good performance.

There are some obvious safeguards that must be noted while using the critical directive. The block of instructions must represent a structured block, i.e., no jumps are permitted into or out of the block. It is easy to see that the former would result in non-critical access and the latter in an unreleased lock, which could cause the threads to wait indefinitely.

Often, a critical section consists simply of an update to a single memory location, for example, incrementing or adding to an integer. OpenMP provides another directive, atomic, for such atomic updates to memory locations. The atomic directive specifies that the memory location update in the following instruction should be performed as an atomic operation. The update instruction can be one of the following forms:

```
x binary_operation = expr
x++
++x
x--
--x
```

Here, expr is a scalar expression that does not include a reference to x, x itself is an lvalue of scalar type, and binary_operation is one of {+, *, -, /, &, '||, <<, >>,}. It is important to note that the atomic directive only atomizes the load and store of the scalar variable. The evaluation of the expression is not atomic. Care must be taken to ensure that there are no race conditions hidden therein. This also explains why the expr term in the atomic directive cannot contain the updated variable itself. All atomic directives can be replaced by critical directives provided they have the same name. However, the availability of atomic hardware instructions may optimize the performance of the program, compared to translation to critical directives.

In-Order Execution: The ordered Directive

In many circumstances, it is necessary to execute a segment of a parallel loop in the order in which the serial version would execute it. For example, consider a for loop in which, at some point, we compute the cumulative sum in array cumul_sum of a list stored in array list. The array cumul_sum can be computed using a for loop over index i serially by executing cumul_sum[i] = cumul_sum[i-1] + list[i]. When executing this for loop across threads, it is important to note that cumul_sum[i] can be computed only after cumul_sum[i-1] has been computed. Therefore, the statement would have to executed within an ordered block.

The syntax of the ordered directive is as follows:

```
#pragma omp ordered
    structured block
```

Since the ordered directive refers to the in-order execution of a for loop, it must be within the scope of a for or parallel for directive. Furthermore, the for or parallel for directive must have the ordered clause specified to indicate that the loop contains an ordered block.

Example 7.17 Computing the cumulative sum of a list using the ordered directive

As we have just seen, to compute the cumulative sum of i numbers of a list, we can add the current number to the cumulative sum of i-1 numbers of the list. This loop must, however, be executed in order. Furthermore, the cumulative sum of the first element is simply the element itself. We can therefore write the following code segment using the ordered directive.

```
cumul_sum[0] = list[0];
#pragma omp parallel for private (i) \
            shared (cumul_sum, list, n) ordered
for (i = 1; i < n; i++)
{
    /* other processing on list[i] if needed */

    #pragma omp ordered
    {
         cumul_sum[i] = cumul_sum[i-1] + list[i];
     }
 }
```

■

It is important to note that the ordered directive represents an ordered serialization point in the program. Only a single thread can enter an ordered block when all prior threads (as determined by loop indices) have exited. Therefore, if large portions of a loop are enclosed in ordered directives, corresponding speedups suffer. In the above example, the parallel formulation is expected to be no faster than the serial formulation unless there is significant processing associated with list[i] outside the ordered directive. A single for directive is constrained to have only one ordered block in it.

Memory Consistency: The flush Directive

The flush directive provides a mechanism for making memory consistent across threads. While it would appear that such a directive is superfluous for shared address space machines, it is important to note that variables may often be assigned to registers and register-allocated variables may be inconsistent. In such cases, the flush directive provides a memory fence by forcing a variable to be written to or read from the memory system. All write operations to shared variables must be committed to memory at a flush and all references to shared variables after a fence must be satisfied from the memory. Since private variables are relevant only to a single thread, the flush directive applies only to shared variables.

The syntax of the flush directive is as follows:

```
#pragma omp flush[(list)]
```

The optional list specifies the variables that need to be flushed. The default is that all shared variables are flushed.

Several OpenMP directives have an implicit flush. Specifically, a flush is implied at a barrier, at the entry and exit of critical, ordered, parallel, parallel for, and parallel sections blocks and at the exit of for, sections, and single blocks. A flush is not implied if a nowait clause is present. It is also not implied at the entry of for, sections, and single blocks and at entry or exit of a master block.

7.10.4 Data Handling in OpenMP

One of the critical factors influencing program performance is the manipulation of data by threads. We have briefly discussed OpenMP support for various data classes such as private, shared, firstprivate, and lastprivate. We now examine these in greater detail, with a view to understanding how these classes should be used. We identify the following heuristics to guide the process:

- If a thread initializes and uses a variable (such as loop indices) and no other thread accesses the data, then a local copy of the variable should be made for the thread. Such data should be specified as private.
- If a thread repeatedly reads a variable that has been initialized earlier in the program, it is beneficial to make a copy of the variable and inherit the value at the time of thread creation. This way, when a thread is scheduled on the processor, the data can reside at the same processor (in its cache if possible) and accesses will not result in interprocessor communication. Such data should be specified as firstprivate.
- If multiple threads manipulate a single piece of data, one must explore ways of breaking these manipulations into local operations followed by a single global operation. For example, if multiple threads keep a count of a certain event, it is beneficial to keep local counts and to subsequently accrue it using a single summation at the end of the parallel block. Such operations are supported by the reduction clause.
- If multiple threads manipulate different parts of a large data structure, the programmer should explore ways of breaking it into smaller data structures and making them private to the thread manipulating them.
- After all the above techniques have been explored and exhausted, remaining data items may be shared among various threads using the clause shared.

In addition to private, shared, firstprivate, and lastprivate, OpenMP supports one additional data class called threadprivate.

The threadprivate and copyin Directives Often, it is useful to make a set of objects locally available to a thread in such a way that these objects persist through parallel and serial blocks provided the number of threads remains the same. In contrast to private variables, these variables are useful for maintaining persistent objects across parallel regions, which would otherwise have to be copied into the master thread's data space and reinitialized at the next parallel block. This class of variables is supported in OpenMP using the threadprivate directive. The syntax of the directive is as follows:

```
#pragma omp threadprivate(variable_list)
```

This directive implies that all variables in variable_list are local to each thread and are initialized once before they are accessed in a parallel region. Furthermore, these variables persist across different parallel regions provided dynamic adjustment of the number of threads is disabled and the number of threads is the same.

Similar to firstprivate, OpenMP provides a mechanism for assigning the same value to threadprivate variables across all threads in a parallel region. The syntax of the clause, which can be used with parallel directives, is copyin(variable_list).

7.10.5 OpenMP Library Functions

In addition to directives, OpenMP also supports a number of functions that allow a programmer to control the execution of threaded programs. As we shall notice, these functions are similar to corresponding Pthreads functions; however, they are generally at a higher level of abstraction, making them easier to use.

Controlling Number of Threads and Processors

The following OpenMP functions relate to the concurrency and number of processors used by a threaded program:

```
#include <omp.h>

void omp_set_num_threads (int num_threads);
int omp_get_num_threads ();
int omp_get_max_threads ();
int omp_get_thread_num ();
int omp_get_num_procs ();
int omp_in_parallel();
```

The function omp_set_num_threads sets the default number of threads that will be created on encountering the nextparallel directive provided the num_threads clause is not used in theparallel directive. This function must be called outsid the scope of a parallel region and dynamic adjustment of threads must be enabled (using either the OMP_DYNAMIC environment variable discussed inSection 7.10.6 or the omp_set_dynamic library function).

The omp_get_num_threads function returns the number of threads participating in a team. It binds to the closest parallel directive and in the absence of a parallel directive, returns 1 (for master thread). The omp_get_max_threads function returns the maximum number of threads that could possibly be created by a parallel directive encountered, which does not have anum_threads clause. The omp_get_thread_num returns a unique thread i.d. for each thread in a team. This integer lies between 0 (for the master thread) andomp get_num_threads() -1. The omp_get_num_procs function returns the number of processors that are available to execute the threaded program at that point. Finally, the function omp_in_parallel returns a non-zero value if called from within the scope of a parallel region, and zero otherwise.

Controlling and Monitoring Thread Creation

The following OpenMP functions allow a programmer to set and monitor thread creation:

```
#include <omp.h>

void omp_set_dynamic (int dynamic_threads);
int omp_get_dynamic ();
void omp_set_nested (int nested);
int omp_get_nested ();
```

The omp_set_dynamic function allows the programmer to dynamically alter the number of threads created on encountering a parallel region. If the value dynamic_threads evaluates to zero, dynamic adjustment is disabled, otherwise it is enabled. The function must be called outside the scope of a parallel region. The corresponding state, i.e., whether dynamic adjustment is enabled or disabled, can be queried using the function omp_get_dynamic, which returns a non-zero value if dynamic adjustment is enabled, and zero otherwise.

The omp_set_nested enables nested parallelism if the value of its argument,nested, is non-zero, and disables it otherwise. When nested parallelism is disabled, any nested parallel regions subsequently encountered are serialized. The state of nested parallelism can be queried using the omp_get_nested function, which returns a non-zero value if nested parallelism is enabled, and zero otherwise.

Mutual Exclusion

While OpenMP provides support for critical sections and atomic updates, there are situations where it is more convenient to use an explicit lock. For such programs, OpenMP provides functions for initializing, locking, unlocking, and discarding locks. The lock data structure in OpenMP is of type omp_lock_t. The following functions are defined:

```
#include <omp.h>

void omp_init_lock (omp_lock_t *lock);
void omp_destroy_lock (omp_lock_t *lock);
void omp_set_lock (omp_lock_t *lock);
void omp_unset_lock (omp_lock_t *lock);
int omp_test_lock (omp_lock_t *lock);
```

Before a lock can be used, it must be initialized. This is done using the omp_init_lock function. When a lock is no longer needed, it must be discarded using the function omp_destroy_lock. It is illegal to initialize a previously initialized lock and destroy an uninitialized lock. Once a lock has been initialized, it can be locked and unlocked using the functions omp_set_lock and omp_unset_lock. On locking a previously unlocked lock, a thread gets exclusive access to the lock. All other threads must wait on this lock when they attempt an omp_set_lock. Only a thread owning a lock can unlock it. The result of a thread attempting to unlock a lock owned by another thread is undefined. Both of these operations are illegal prior to initialization or after the destruction of a lock. The function omp_test_lock can be used to attempt to set a lock. If the function returns a non-zero value, the lock has been successfully set, otherwise the lock is currently owned by another thread.

Similar to recursive mutexes in Pthreads, OpenMP also supports nestable locks that can be locked multiple times by the same thread. The lock object in this case is omp_nest_lock_t and the corresponding functions for handling a nested lock are:

```
#include <omp.h>

void omp_init_nest_lock (omp_nest_lock_t *lock);
void omp_destroy_nest_lock (omp_nest_lock_t *lock);
void omp_set_nest_lock (omp_nest_lock_t *lock);
void omp_unset_nest_lock (omp_nest_lock_t *lock);
int omp_test_nest_lock (omp_nest_lock_t *lock);
```

The semantics of these functions are similar to corresponding functions for simple locks. Notice that all of these functions have directly corresponding mutex calls in Pthreads.

7.10.6 Environment Variables in OpenMP

OpenMP provides additional environment variables that help control execution of parallel programs. These environment variables include the following.

OMP_NUM_THREADS This environment variable specifies the default number of threads created upon entering a parallel region. The number of threads can be changed using either the omp_set_num_threads function or the num_threads clause in the parallel directive. Note that the number of threads can be changed dynamically only if the variable OMP_SET_DYNAMIC is set to TRUE or if the function omp_set_dynamic has been called with a non-zero argument. For example, the following command, when typed into csh prior to execution of the program, sets the default number of threads to 8.

```
setenv OMP_NUM_THREADS 8
```

OMP_DYNAMIC This variable, when set to TRUE, allows the number of threads to be controlled at runtime using the omp_set_num threads function or the num_threads clause. Dynamic control of number of threads can be disabled by calling the omp_set_dynamic function with a zero argument.

OMP_NESTED This variable, when set to TRUE, enables nested parallelism, unless it is disabled by calling the omp_set_nested function with a zero argument.

OMP_SCHEDULE This environment variable controls the assignment of iteration spaces associated with for directives that use the runtime scheduling class. The variable can take values static, dynamic, and guided along with optional chunk size. For example, the following assignment:

```
setenv OMP_SCHEDULE "static,4"
```

specifies that by default, all for directives use static scheduling with a chunk size of 4. Other examples of assignments include:

```
setenv OMP_SCHEDULE "dynamic"
```

```
2   setenv OMP_SCHEDULE "guided"
```

In each of these cases, a default chunk size of 1 is used.

7.10.7 Explicit Threads versus OpenMP Based Programming

OpenMP provides a layer on top of native threads to facilitate a variety of thread-related tasks. Using directives provided by OpenMP, a programmer is rid of the tasks of initializing attributes objects, setting up arguments to threads, partitioning iteration spaces, etc. This convenience is especially useful when the underlying problem has a static and/or regular task graph. The overheads associated with automated generation of threaded code from directives have been shown to be minimal in the context of a variety of applications.

However, there are some drawbacks to using directives as well. An artifact of explicit threading is that data exchange is more apparent. This helps in alleviating some of the overheads from data movement, false sharing, and contention. Explicit threading also provides a richer API in the form of condition waits, locks of different types, and increased flexibility for building composite synchronization operations as illustrated in Section 7.8. Finally, since explicit threading is used more widely than OpenMP, tools and support for Pthreads programs is easier to find.

A programmer must weigh all these considerations before deciding on an API for programming.

[Team LiB]   PREVIOUS NEXT