USING OPENMP IN A PARALLEL COMPUTING COURSE

Thomas B. Gendreau
Computer Science Department
University of Wisconsin - La Crosse
La Crosse, WI 54601
gendreau@cs.uwlax.edu

Abstract

Multicore processors in laptop and desktop machines make small scale parallel processing available to most undergraduate computer science students. OpenMP is a directive based API that provides a simply way to introduce parallel computing ideas to undergraduate computer science students. OpenMP fills a gap between having the compiler discover implicit parallelism and having a programmer handle all the details of thread creation and synchronization in a parallel application. Through the use of constructs specified by compiler directives a programmer can indicate blocks of code that can execute in parallel. This paper provides an overview of OpenMP features and discusses its use in a parallel computing class.

1. Introduction

At the University of Wisconsin - La Crosse CS 470/570: Parallel and Distributed Computing is a first course in parallel processing. The course illustrates parallel and distributed processing issues with a variety of APIs including Java Threads, pthreads and MPI. A new addition to the course is the use of a directive based API called OpenMP. OpenMP fills a gap between having the compiler discover implicit parallelism and having a programmer handle all the details of thread creation and synchronization in a parallel application.

OpenMP [1][2][3] is a set of compiler directives and library routines to support parallel programming in a shared memory architecture. OpenMP can be used with C, C++ and ForTran. With OpenMP directives a programmer can specify code that can be executed in parallel and where synchronization is required but the compiler handles the details of creating threads and synchronizing threads. The format (in C/C++) of an OpenMP directive is *#pragma omp construct [clauses]*, where construct can be one of the many constructs supported by OpenMP. A directive is associated with a block of code that has one entry point and one exit point. In a program that uses OpenMP there is a master thread that runs for the life of the program. When a parallel block entry point is reached, additional threads are created. The exit point of a block is an implicit synchronization point for the master thread and the additional threads created for the block. After the synchronization at the exit point is completed the master thread continues with the computation and the other threads end.

2. A Simple Example

As a simple example consider the following code segment. The code segment finds the (approximate) area under the curve $f(x) = x^2$ between 0 and 1 by summing the areas of rectangles under the curve. The master thread executes the first two lines of code. When the compiler sees the *#pragma* directive it generates code to execute the for loop with multiple threads. The threads synchronize when all iterations of the for loop are finished. The master thread executes the last statement in the example.

```
numIntervals = atoi(argv[1]);
 width = 1.0 / numIntervals;
 #pragma omp parallel for default(shared) private(i, midpoint) num_threads(4) \
   reduction(+:sum)
      for(i = 0; i < numIntervals; i++) {
           midpoint = (i + 0.5) * width;
           sum = sum +midpoint * midpoint;
      }

 area = width*sum;
```

The directive *#pragma omp parallel for* indicates that each iteration of the for-loop can be executed independently. Like other C preprocessor commands if directives require

more that one line a "\" character must appear at the end of all but the last line used by the directive. There are many different clauses that can be used with various constructs. The clauses in this example are shown below with a brief explanation. Not all clauses are required.

default(shared) **:** Variables in the block are assumed to be shared between threads

private(i, midpoint) **:** Variables i and midpoint have local copies for each thread. All variables, shared or private, must be declared before the parallel block.

num_threads(4) : 4 threads should be used for the block.

reduction(+:sum) **:** There will be a local variable sum for each thread. When the threads complete, the local sum values should be added to produce a single sum value.

3. Parallel Directives in OpenMP

The first step in learning OpenMP is to learn some constructs that can be used to specify parallelism. The section covers some of the constructs used to specify parallelism. More details can be found in the references.

Three of the constructs that indicate code that can be executed in parallel are *#pragma omp parallel*, *#pragam omp sections*, *#pragma omp for*. Any code that should be executed in parallel must be in a structured block associated with a *#pragma omp parallel* directive**.** The other directives shown above can be separate blocks in a parallel block or can be specified by combining the parallel construct with the other constructs. For example the code segment in the previous section combined the parallel and for constructs.

The simplest way to specify a block of parallel code is to use *#pragma omp parallel.* When no other constructs are used with the parallel construct, the code in the structured block is replicated once for each thread that is created. The number of threads created depends on the number of processors or cores or can be specified with the *num_threads* clause.

Considered a simple data parallel program that applies a function f to each of n files. Suppose the result of the function depends only on the file to which it is applied and the result is stored in an array. The following code segment creates one thread for each file and applies the function to each file.

```
FILE *theFiles[MAXFILES];
RESULT_TYPE results[MAXFILES];

...

#pragma omp parallel default(shared) private(i) num_threads(n)
{
        i = omp_get_thread_num();
        results[i] = f(theFiles[i]);
}
```

The function *omp_get_thread_num()* returns the thread number associated with each thread created in the structured block. The master thread is given the number 0 and the other threads are given numbers 1 through n-1. It is a common technique to use the thread number to identify the work a thread should do. Indicating that i is private means that each thread has its own private copy of i.

The parallel construct makes it easy to parallelize code but the above example may create too much parallelism for a machine with just a couple of cores. The code shown below uses the *sections* clause to process the files one pair at a time.

```
for (i = 0; i <n; i=i+2) {
        #pragma omp parallel sections default(shared)
        {
                #pragma omp section
                {
                        result[i] =f(theFiles[i]);
                }

                #pragma omp section
                {
                        if ( i+1 < n) result[i+1] =f(theFiles[i+1]);
                }
        }
}
```

Each *sections* block will include 2 or more *section* blocks. The master thread will do the work of one *section* block and new threads are created for the other *section* blocks. Since the above *sections* block has two *section* blocks, the master thread will process one section block and a new thread will be created for the other *section* block. Since there is an implicit synchronization point at the end of the *sections* block a new iteration of the for loop can only begin after the work of both *section* blocks in the previous iteration is complete.

A sequential solution to the simple problem we have been discussing would include a loop as shown below.

```
for (k =0; k <n; k++)
        result[k] = f(theFiles[k]);
```

Since the iterations of the loop are independent, each iteration can be executed in parallel. The code shown below uses the *for* construct to indicate that the iterations of the loop can be done in parallel.

```
#pragma omp parallel for
{
        for (k =0; k <n; k++)
                result[k] = f(theFiles[k]);
}
```

The number of threads created will depend on the number of processors or cores. On a dual core processor the master thread will do half the iterations and a new thread will be created to do the other half of the iterations. The code matches the number of threads used to the number of processors or cores in the machines. It will not create as much parallelism as the version that created one thread for each file and it will match the parallelism to the number of processors or cores better than the version that used the sections construct.

4. Synchronization Directives in OpenMP

In the examples shown in the previous section the only synchronization the code segments require is the implicit synchronization at the end of each structured block. This synchronization occurs because the master thread will not proceed to execute the code following a structured block until all threads created for that structured block have completed (the *nowait* clause can be used to override this feature). Many applications will require threads to synchronize during execution of the threads. Two OpenMP directives that can be used for synchronization are *#pragma omp barrier* and *#pragma omp critical*.

The directive *#pragma omp barrier* implements classic barrier synchronization. When threads reach the barrier they will be blocked until all threads have reached the barrier.

The code shown below is a parallel implementation that merges two sorted halves of an array [4]. The code assumes each half is already sorted. Each thread finds the proper place for one element in the array. The proper place for an element is determined by each thread by taking into account the position of the element in its half of the array and the place the element would be inserted in the sorted sequence of the other half of the array. When all threads have found the proper place for their element, the array is updated. Since this update changes the content of the shared array, the update should not happen until all threads have found the proper place for their element. In the code shown below

this synchronization requirement is enforced by the *#pramga omp barrier* directive listed immediately before the update statement.

```
#pragma omp parallel default(shared) private(i, myId, x, high, low)  \
  num_threads(size)
{
        myId = omp_get_thread_num();
        if (myId < size2) {
                low = size2;
                high = size-1;
        }
        else {
                low = 0;
                high = size2-1;
        }
        x = nums[myId];
        do {
                i = (low+high)/2;
                if (x < nums[i])
                        high = i-1;
                else
                        low = i+1;
        } while (low <= high);
        #pragma omp barrier
        nums[high+myId-(size2-1)] = x;
}
```

Many parallel algorithms have critical sections of code that should be executed by only one thread at a time. With the *#pragma omp critical* directive a programmer can specify critical section code. Suppose we want a program that finds the largest value in n files of integers and we want to find the largest value in all files. One way to do this with OpenMP is shown below.

```
FILE *theFiles[MAXFILES];
int fileMax[MAXFILES];
int max = INT_MIN;

...

#pragma omp parallel default(shared) private(i) num_threads(n)
{
        i = omp_get_thread_num();
        fileMax[i] = findMax(theFiles[i]);
        #pragma omp critical
        {
                if (fileMax[i] > max)
                        max = fileMax[i];
        }
}
```

Since max is a shared variable that can be updated, the code for reading and updating max must be in a critical section.

Other OpenMP synchronization directives include *#pragma omp master* and *#pragma omp single*. A block of code associated with the *master* directive will only be executed by the master thread. A block of code associated with the *#pragma omp single* directive will only be executed by one of the threads in the parallel section.

5. OpenMP Functions

In addition to compiler directives OpenMP includes a library of functions. The function signatures of a representative list of functions are shown below.

```
int omp_get_thread_num()
int omp_get_num_threads()
int omp_get_num_procs()
void omp_init_lock(omp_lock_t)
void  omp_set_lock(omp_lock_t))
void  omp_unset_lock(omp_lock_t))
void omp_destroy_lock(omp_lock_t))
double omp_get_wtime()
double omp_get_wtick()
```

The first three functions in the list allow a program to find the value of some property about the environment in which the program is running. We have already seen that the function *omp_get_thread_num* returns the thread number associated with a thread. This will always be a value between 0 and n-1 when n threads are running. The master thread

always has thread number 0. The function *omp_get_num_threads* returns the number of threads that are currently executing. The function *omp_get_num_procs* returns the number of processors or cores available in the machine on which the program is executing.

The four lock methods allow a program to initialize, use and destroy an *omp_lock_t*. The *omp_set_lock* function allows a thread to acquire a lock. If another thread has the lock the thread executing the *omp_set_lock* function will be blocked until the thread that has the lock executes the *omp_unset_lock* function.

The *omp_get_wtime* function returns wall clock time. Calls to this function before and after executing a parallel block of code can be used to create performance data. The accuracy of the values returned for this function varies from system to system. The function *omp_get_wtick* can be used to find the precision of the clock.

6. Teaching with OpenMP

One of the advantages of using OpenMP is that it is easy to get students to write parallel programs without the complexities of pthreads. The features of OpenMP can be illustrated to the students with a simple example. Consider a problem with three arrays of integers M, A and B. Assume A and B have each been assigned size values. The problem is to assign to each M[i] the maximum value found in A[i] and B[i]. A sequential solution to the problem is shown below.

```
for (i = 0; i < size; i++) {
        if (A[i] > B[i])
                M[i] = A[i];
        else
                M[i] = B[i];
}
```

When discussing how to parallelize this code the students can observe that the iterations of the loop are independent and the instructor can show the variety of ways OpenMP can take advantage of this fact.

The first solution makes use of a data parallel approach to the problem. The OpenMP *parallel* construct can be used to create size threads and to assign each thread responsibility for one slot in each array. The code below shows this solution.

```
#pragma omp parallel default(shared) private(i) num_threads(size)
{
        i = omp_get_thread_num();
        if (A[i] > B[i])
                M[i] = A[i];
        else
                M[i] = B[i];
}
```

With the code segment the instructor can introduce the students to the format of OpenMP directives, constructs, clauses and functions. The issue of shared and private variables can be illustrated by looking at the results of the program when the *private(i)* clause is omitted. The common technique of using the thread number to determine the work assigned to each thread can also be illustrated with this simple example.

The solution shown above uses the maximum parallelism possible and can be used to discuss the problem of matching the amount of parallelism implemented by a program to the architecture on which the program will execute. This can lead to a discussion of a solution based on task decomposition and the OpenMP *sections* construct. The solution shown below uses a *sections* construct with two *section* blocks (or two tasks). It provides a limited use of parallelism that could match an architecture with a couple processors.

```
#pragma omp parallel sections default(shared) private(i)
{
        #pragma omp section
        {
                for (i = 0; i < size/2; i++)
                if (A[i] > B[i])
                        M[i] = A[i];
                else
                        M[i] = B[i];
        }
        #pragma omp section
        {
                for (i = size/2; i < size; i++)
                if (A[i] > B[i])
                        M[i] = A[i];
                else
                        M[i] = B[i];
        }
}
```

The solution uses two threads to assign the proper values to the elements in array M. One thread is responsible for the first half of the elements and the other thread is responsible

for the second half of the elements. Discussion with students can include the relative merits of the data parallel approach versus the task decomposition approach.

A third solution that can be discussed is based on the *for* construct. The solution is shown below

```
#pragma omp parallel for
for (i = 0; i < size; i++) {
        if (A[i] > B[i])
                M[i] = A[i];
        else
                M[i] = B[i];
}
```

This solution is a variation on a task decomposition approach. The number of threads created will depend on the number of processors or cores on the machine. Each thread will be assigned the work of its share of the iterations. Since the number of threads depends on the architecture this solution is more portable than the solution that breaks the problem into exactly two threads.

One more requirement can be added to the problem to illustrate another feature of OpenMP. If in addition to assigning the correct values to the array M the problem also requires that the maximum value among all the values in the arrays A and B must be found. A solution to this additional requirement can illustrate the use of the *critical* construct.

The sequential solution with this additional requirement is shown below.

```
for (i = 0; i < size; i++) {
        if (A[i] > B[i])
                M[i] = A[i];
        else
                M[i] = B[i];
        if (M[i] > max)
                max = M[i];
```

In the discussion of this problem students can observe that the assignment of a value to max is not independent of the iteration. This can lead to a discussion of critical sections and how to implement them in OpenMP. The code below uses the directive *#pragma omp critical* to indicate the critical section of code.

```
#pragma omp parallel default(shared) private(i) num_threads(size)
{
        i = omp_get_thread_num();
        if (A[i] > B[i])
                M[i] = A[i];
        else
                M[i] = B[i];
        #pragma omp critical
        {
                if (M[i] > max)
                        max = M[i];
        }
}
```

While size threads will be created in the parallel block only one thread at a time will be able to execute the if statement.

7. Conclusion

OpenMP provides a simple set of directives and functions for parallel programming in a shared memory environment. Students can write parallel programs that run on their multicore laptops and desktops without the development overhead of pthreads. OpenMP is simple enough it could be used in introductory courses that use C or C++ or in advanced courses focused exclusively on parallel computing.

References

1. Grama, Ananth, Gupta, Anshul, Karypis, George and Kumar, Vipin (2003). *Introduction to Parallel Computing*, 2nd Edition. Addison-Wesley.

2. computing.llnl.gov/tutorials/openMP/

3. openmp.org

4. Quinn, Michael J. (1994). *Parallel Computing*. McGraw Hill.