An Introduction to OpenMP

Richard Fujimoto

October 2014
(updated October 2017)

This primer gives a basic "how to" tutorial to create and run OpenMP programs. Parallel programs in OpenMP (and many other parallel programming systems) are based on a construct called "threads." When you execute a conventional C program sequentially on a single machine, that program executes as a single thread. Parallel programs consist of a collection of threads, each of which may run in parallel with other threads on different processors. Operationally a thread includes storage for the machine instructions making up the program itself, a program counter indicating the address of the instruction in the program to be executed next, a stack to provide storage for local variables, function arguments, return addresses, etc., and various other information managed by the operating system that are used by the program as it executes. The operating system manages the threads making up the parallel program and allocates resources (processors, memory) to execute them.

OpenMP provides basic constructs to (1) create multiple threads of execution that can execute concurrently, (2) define shared variables that allow the threads to exchange information, and (3) provides mechanisms to allow threads to synchronize with each other in order to control the order that operations are performed. These are the basic ingredients needed to write parallel programs to execute on shared memory multiprocessor systems in general.

1. Getting Started: Hello World

Let's start with writing a simple parallel "hello world" program and walk through executing it on jinx. This will introduce the basic constructs needed to create threads. The steps below involve creating and running a simple multi-threaded program. The program executes in three steps (see Figure 1):

1. The `main()` program begins executing as a sequential program on a single processor, just like any other C program.
2. A set of threads are created by invoking the "omp parallel { … }" primitive. This creates several versions (threads) of the code block enclosed in the { … }, allowing each to execute on a separate processor. Here, we will create multiple threads, with each one simply printing "hello world" and then terminating.
3. When all of the parallel threads complete executing their code block (the code within the braces of the omp parallel primitive), the main (sequential) program resumes execution at the statement following the braces.

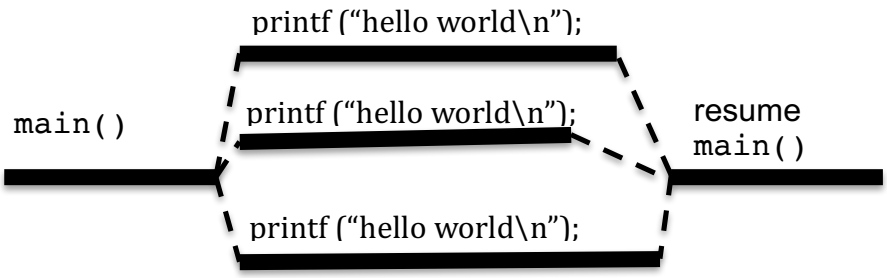

Figure 1. Creating threads for "hello world" program.

1.1 Step 1: Develop the C Program

In general, you need to first develop a parallelization strategy before you code your algorithm, but let's assume this has been done. We will start with the original (sequential) C program for "hello world," and augment it for parallel execution, i.e., we will create a set of independently executing threads.

Open MP uses compiler "pragmas" to instruct the compiler to augment your program with various primitives to enable parallel execution. A few key things you need to know:

1. You need to add the line `#include <omp.h>` at the beginning of your program to give access to the OpenMP libraries.
2. Use the function `omp_set_num_threads(k)` to specify that you want k threads to be created that will execute concurrently.
3. Insert `#pragma omp parallel { … }` to execute the code within the braces, each as a separate thread. Each thread will execute concurrently with the other threads, possibly on different processors.
4. Each thread is assigned a unique ID so that it can use to distinguish itself from other threads. A thread can determine its ID by calling the function `omp_get_thread_num()`. Thread IDs are numbered 0, 1, 2, etc.

The parallel "hello world" program is below. This code will spawn two threads and execute them, possibly on different processors. Once all of the threads have completed execution, the program resumes execution after the parallel (enclosed in braces of the `#pragma omp parallel)` primitive. Create a file, say hello.c that holds this code.

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

int main (void)
{
    omp_set_num_threads(2);

    #pragma omp parallel
    {
        int ID=omp_get_thread_num();
        int num_threads=omp_get_num_threads();
        printf ("Hello(%d of %d) ", ID, num_threads-1);
        printf (" World(%d of %d)\n", ID, num_threads-1);
    }
```

```
    printf ("Done\n");
}
```

1.2 Step 2: Run Your Program

Log into the compute cluster on which you will execute your code, transfer your source files to that machine, and compile and link the program. Refer to instructions for your local computer cluster to figure out how to do this. When you run your program it should print to the screen something like this:

```
Hello(0 of 1) World(0 of 1)
Hello(1 of 1) World(1 of 1)
Done
```

The program could also produce the following output because the order of execution of the `printf` statements among the different threads is arbitrary:

```
Hello(1 of 1) Hello(0 of 1) World(0 of 1)
World(1 of 1)
Done
```

In general, if you have a set of threads executing in parallel on different processors, the order in which operations of the various threads are executed may be interleaved in an arbitrary fashion, and so may be different from one run to the next. This can be a bad thing because we usually want the program to always produce the same answers when we run it over and over again. When writing parallel programs you must work out a scheme to ensure you get repeatable results (assuming this is what you want).

2. Shared and Private Variables

The "hello world" program did not involve any communication or interaction among the threads. Typically, threads will interact with each other, e.g., one thread might use results produced by another thread. Shared variables are used to accomplish this in OpenMP.

There are essentially two types of variables accessed by OpenMP programs termed *shared* and *private* variables. Shared variables are those that can be accessed by multiple threads. If a thread modifies a shared variable the new value can then be read by the other threads (who may then modify it again). By default, most variables defined in C programs will be shared variables. One should be very careful when dealing with shared variables that are modified by more than one thread because "race" conditions may arise when the interleaving of the execution of individual statements by different threads is not what one intended; this can lead to unexpected results. In general, one should use synchronization mechanisms (described later) to manage this issue.

Unlike shared variables, the private variables of a thread can only be accessed by that thread. Modifications to private variables cannot be seen by other threads; issues such as race conditions do not arise. When the code for a thread declares a local variable each thread is allocated its own copy of these variables. Local variables declared within a parallel block, or local variables within functions invoked within a parallel block are private. In general, the variables stored in the runtime stack of a thread will be private variables accessible only to that thread.

A parallel program is shown below for computing the dot product of two vectors. Here, the global variables `A, B, C,` and `Result` are shared variables. The local variables

defined within `main()` (`i` and `nthreads`) are also shared variables. The variables defined within the threads defined by the parallel pragma (`i`, `id`, `nthrds`, and `MyC`) are private – a separate version of each is created for each thread created using the parallel pragma. If one or more of the parallel threads called a function, say `foo()` that also declared local variables, these local variables defined within `foo()` would also be private. These variables are stored in the thread's stack and modifications to these variables are only visible within the thread in which it is created.

Notice how the dot product computation is distributed across the different threads. The heart of the sequential computation is a `for` loop that goes through each element of the arrays `A` and `B` and multiplies them together. In the parallel code, the iterations of this `for` loop are distributed, round robin, to the different threads. For example, if there are 4 threads, the first iteration will go on the first thread, the second on the second thread, the third (fourth) iteration on the third (fourth) thread, respectively. The fifth is assigned to the first thread, etc. Equivalently, iterations 0, 4, 8, … are assigned to thread 0, iterations 1, 5, 9, … to thread 1, iterations 2, 6, 10, … to thread 2, and iterations 3, 7, 11, … to thread 4. This is a common, widely used approach to distribute the computation within a loop across multiple threads.

Each thread computes a "local" dot product that is stored in the private variable `MyC`, and after completing the execution of the loop, the computed result for thread i is stored in a global variable C[i]. Note that there is one array element of C for each thread. After the parallel threads all complete execution, the code following the parallel thread code is executed sequentially to sum the values computed by each thread.

```
//
// Parallel Program to compute dot product
//

#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 4
#define  NUM_ELEM 100

// Global Variables
double A[NUM_ELEM], B[NUM_ELEM];  // inputs to computation
double C[NUM_THREADS];            // computed by each thread
double Result;                    // final result

int main (void)
{
        int i, nthreads;          // shared variables

        // initialize arrays
        for (i=0; i<NUM_ELEM; i++) {
             …
}

        omp_set_num_threads(NUM_THREADS);
```

```
        #pragma omp parallel
        {
                int i, id, nthrds;        // private variables
                double MyC;

                id = omp_get_thread_num();
                nthrds = omp_get_num_threads();
                 // nthreads a global; only 1 thread writes to it
                if (id == 0) nthreads = nthrds;
                for (i=id, MyC=0.0;  i<NUM_ELEM;  i+=nthrds) {
                        MyC += A[i]*B[i];
                }
                C[id] = MyC;  // write result from this thread
        }

        // compute the overall result (sequentially)
        for (i=0, Result=0; i<nthreads; i++)  Result += C[i];
        printf ("result = %f\n", Result);
}
```

Finally, it is worth noting that accesses to local variables are usually faster than accesses to shared variables. This is because local variables will typically be stored in the cache memory of the processor executing the code, and will usually remain there so long as the thread keeps accessing that variable. Shared variables will also be stored in cache memory, however, there are more complex activities that go on “behind the scenes” to manage shared variables in caches, particularly when they are modified. This can cause accesses to shared variables to take much longer, causing your program to run more slowly. Therefore, as a general rule you should use local (private) variables wherever possible, and use shared variables only when data needs to be accessed outside the thread. For example, in the code shown above, the variable `MyC` was created as a local, private variable and used by the thread to accumulate the sum. When done accumulating the value, the result is written into the shared variable `C[id]` so the final result produced by the thread can be accessed by other threads. The program could have been written to eliminate `MyC` and use `C[id]` instead, however it likely would run more slowly.

3. Synchronization

There are two main types of synchronization primitives used by parallel programs using the shared memory programming model: barrier synchronizations and critical sections. Each of these are described next.

3.1 Barrier Synchronizations

There are instances where a thread must wait until other threads have “caught up” before advancing forward. OpenMP provides a primitive called a barrier synchronization for this purpose. Consider the code shown below.

```
#pragma omp parallel
```

```
{
     <computation 1>
     # pragma omp barrier
     <computation 2>
}
```

Each of the threads will concurrently execute the statements included in “computation 1.” When a thread executes the barrier pragma, it blocks, and waits until *all* of the threads in the parallel block have also reached the barrier primitive. Once all of the threads have executed the barrier pragma, all are released and continue execution beyond the barrier. This primitive might be needed if, for instance, the threads performing computation 1 collectively produce some result that is used by computation 2. The barrier ensure that computation 2 does not prematurely execute, before the result from computation 1 has been completed.

We saw something like this in the dot product program described earlier. The serial code following the parallel section that sums the values computed by the individual threads must wait until these threads have computed their value and stored the result in `C[]`. In this case, OpenMP automatically forces the serial code following the omp parallel pragma to wait until all of the parallel threads executed by this pragma have completed execution. In effect, OpenMP includes a barrier synchronization primitive at the end of each thread that is executed before the thread terminates.

3.2 Critical Sections

Suppose that one has several threads that are concurrently modifying shared variables. Recall that the order that the individual statements executed by the different threads can be interleaved in arbitrary ways. This can result in unexpected, incorrect results. Worse, the incorrect results may not be repeatable because a subsequent execution of the program may result in a different interleaving of the execution, and produce entirely different results. This is known as a “race” condition. Race conditions, especially those that results in errors infrequently, are among the most difficult programming bugs to detect and fix because they are not easily reproduced.

To combat problems such as these, whenever a thread modifies shared variables, one should ensure that only one thread is modifying these shared variables at a time. This is accomplished by enclosing the block of code into what is called a critical section. OpenMP will ensure that at most one thread is executing statements in a critical section at one time. Specifically, OpenMP defines the primitive “`#pragma omp critical {…}`” where the statements within braces form the critical section. When a thread executes this pragma, one of two things will happen. If no other thread is currently executing statements within the critical section, the thread enters the critical section and begins executing the statements within the braces. On the other hand, if another thread is executing statements within the critical section, then the thread attempting to enter the critical section will block i.e., wait. When a thread leaves the critical section, i.e., completes executing the statements within the braces, OpenMP will check to see if there are other threads waiting to enter the critical section. If there are one or more waiting threads, exactly one thread is allowed to enter the critical section. The selection

of which thread is allowed to enter the critical section depends on the operating system, and should be considered to be arbitrary.

There is a certain "cost" (in time) to enter a critical section. Therefore, one should try to minimize how often one enters and leaves critical sections, and unnecessary critical sections should be eliminated. Having said that, race conditions are problematic, and one should not hesitate to use a critical section if it is needed to ensure the code executes reliably.

The following code is a version of the dot product code modified to use a critical section. Here rather than executing the final summation after the parallel threads complete, the summation is performed within each thread. The array `C[]` has been eliminated as well as the for loop following the parallel section. After computing its local dot product, each thread adds its locally computed value into the total sum (the variable `Result`) before terminating. The critical section includes the single statement "`Result+=MyC`". A critical section is needed here because `Result` is a shared variable that is modified by each thread.

```
//
// Revised Parallel Program to compute dot product
// Includes the final summation as part of the parallel threads
//

#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 4
#define  NUM_ELEM 100

// Global Variables
double A[NUM_ELEM], B[NUM_ELEM];  // inputs to computation
double Result=0.0;                // final result

int main (void)
{
        int i, nthreads;          // shared variables

        // initialize arrays
        for (i=0; i<NUM_ELEM; i++) {
                A[i] = (double) (i);
                B[i] = (double) (i);
        }

        omp_set_num_threads(NUM_THREADS);

        #pragma omp parallel
        {
                int i, id, nthrds;        // private variables
                double MyC;
```

```
                id = omp_get_thread_num();
                nthrds = omp_get_num_threads();
                 // nthreads a global; only 1 thread writes to it
                if (id == 0) nthreads = nthrds;
                for (i=id, MyC=0.0;  i<NUM_ELEM;  i+=nthrds) {
                        MyC += A[i]*B[i];
                }

                 // critical section to sum up results
                 #pragma omp critical
                 {
                     Result += MyC;
                 }
        }

         printf ("result = %f\n", Result);
}
```

Notice that the critical section was placed outside the `for` loop. One could have written the code so each iteration of the `for` loop updated the `Result` variable (and `MyC` could be eliminated), however this would be rather inefficient. Recall that critical sections are time consuming to execute, and they force serial execution of threads. One should write one's code to minimize the use of critical sections as much as possible.

4. Parallel For Loops and Reductions

The example presented in the previous section had each thread compute a value, and then aggregated the results by performing a computation (in this case addition) over all of the results computed by the individual threads. Suppose we have an operator (in the above, addition) to be performed over a set of data elements that are stored in different threads. Further, assume the operator is associative, i.e., the order in which the operator is invoked does not matter (e.g., (A+B)+C = A+(B+C) for addition). This type of computation is called a reduction, and it frequently arises in parallel computing. A reduction is a parallel computing primitive that executes an associative operator over all of the elements of a data set. It occurs so often that hardware is often optimized to execute the reduction operation efficiently.

OpenMP provides a primitive for executing reductions. Before discussing this we need to introduce the OpenMP "for" construct. The "#pragma omp for" construct tells OMP that the individual iterations of the loop that follows is to be split up among the tasks that are available to execute the program. This construct reduces some of the extra code required to set up separate iterations to execute in different threads.

Reductions build upon the OpenMP for pragma. It is illustrated in the code shown below. Specifically, "`#pragma omp for reduction (+:Result)`" within the parallel construct will execute the individual iterations of the for loop concurrently. It will also create a local copy of the `Result` variable, and initialize it (the initialization value depends on the reduction operator; for addition, it is initialized to zero). When the `for`

loop executes, updates to `Result` are performed on the local copy. Finally, the local copies will be reduced into a single value that is stored into the shared `Result` variable.

```
//
// Revised Parallel Program to compute dot product using a reduction
//

#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 4
#define  NUM_ELEM 100

// Global Variables
double A[NUM_ELEM], B[NUM_ELEM];  // inputs to computation
double Result=0.0;                // final result

int main (void)
{
        int i, nthreads;          // shared variables

        // initialize arrays
        for (i=0; i<NUM_ELEM; i++) {
                A[i] = (double) (i);
                B[i] = (double) (i);
        }

        omp_set_num_threads(NUM_THREADS);
        #pragma omp parallel
        {
                int i;        // private variables

                 #pragma omp for reduction (+:Result)
                for (i=0;  i<NUM_ELEM;  i++) {
                        Result += A[i]*B[i];
                }
         printf ("result = %f\n", Result);
}
```

This program executes a construct often referred to as a parallel for loop, meaning each iteration of the loop can be viewed as a separate thread that executes concurrent with the other threads (iterations). Notice that this code is simpler than the original code that was first presented to manage the parallel execution in that it eliminates the effort needed to divide up the iterations among the various threads. The compiler implements this task.

5. Summary

This tutorial gave an introduction to OpenMP. As can be seen, relatively few constructs are needed to get started in writing parallel programs:

- `#pragma omp parallel` is used to create multiple threads that can execute concurrently.
- Most variables are shared among the threads, however, local variables declared within each of the threads will be private.
- Function calls such as `omp_get_num_threads()` and `omp_get_thread_num()` provide information such as the number of executing threads and the ID of an executing thread; `omp_set_num_threads()` can be used to set the number of threads.
- `#pragma omp barrier` and `#pragma omp critical` provide primitives for synchronization.

The above primitives should be enough to get you started. The `for` loop and reduction constructs may be useful and can simplify the code that is developed.

There is much information on OpenMP available on the web. The OpenMP web site is http://openmp.org/wp/ and the tutorial at https://computing.llnl.gov/tutorials/openMP/ may be of particular interest.