Parallel Programming Using OpenMP

by Tiago Sommer Damasceno

Introduction

This module teaches about the basic concepts of parallel programming that uses a multi-threading API called OpenMP.

Requirements

- Knowledge in C Programming Language.
- C compiler that supports OpenMP.
- Unix/Linux terminal knowledge.
- Access to a multi-core computer.

Goals

- Learn basics of OpenMP, such as compiler directives, functions and environment variables;
- Understand how OpenMP works with shared memory.

What is it?

OpenMP is a standard API that can be used in shared memory programs, which are written in Fortran, C, and C++.

- It consists of:
  - Compiler Directives.
  - Functions.
  - Environment Variables.
- Requires a supportive compiler, such as:
  - GNU, IBM, Oracle, Intel, HP, MS, Cray, Absoft Pro Fortran, Portland Group Compilers and Tools, Lahey/Fujitsu Fortran 95, Path Scale.
- The last version, OpenMP 3.0 was release in May, 2008.
- A summary of directives, functions and environment variables can be found at the Petascale Website (www.shodor.org/petascale). or at www.OpenMP.org.

Pros and Cons

Pros:

- No need for major changes in the serial code.
- Portable.
  (Supported by many compilers)
- Has great scalability
  (If implemented correctly and the computer's architecture permits)
- Data decomposition is handled automatically.
- Does not need to deal with message passing.

Cons:

- Requires a compiler that supports OpenMP
- Currently only runs efficiently on shared-memory multiprocessors architectures.
- Scalability limited by the computer's architecture.
- Can not be used on GPU.

Main Components

Directives

- Initializes parallel region.
- Divides the work.
- Synchronizes.
- Data-Sharing Attributes.

Functions

- Defines number of threads.
- Gets thread ID.
- Supports Nested Parallelism.

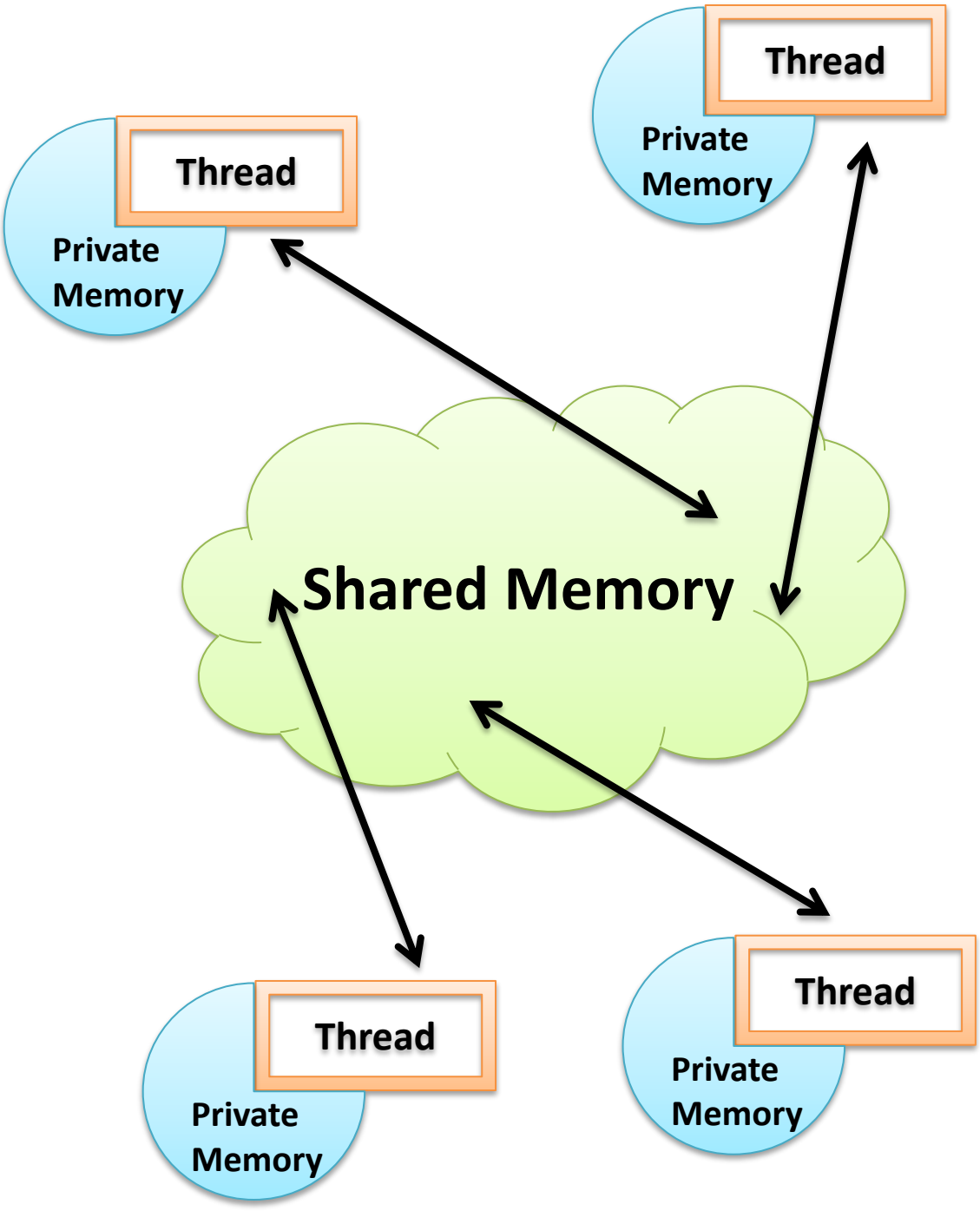
Environment Variables

- Scheduling Type.
- Number of Threads.
- Dynamic Adjustment of Threads.
- Maximum Number of Threads.

✓For a more specific explanation of those three components, go to www.openMP.org and look for OpenMP Specifications

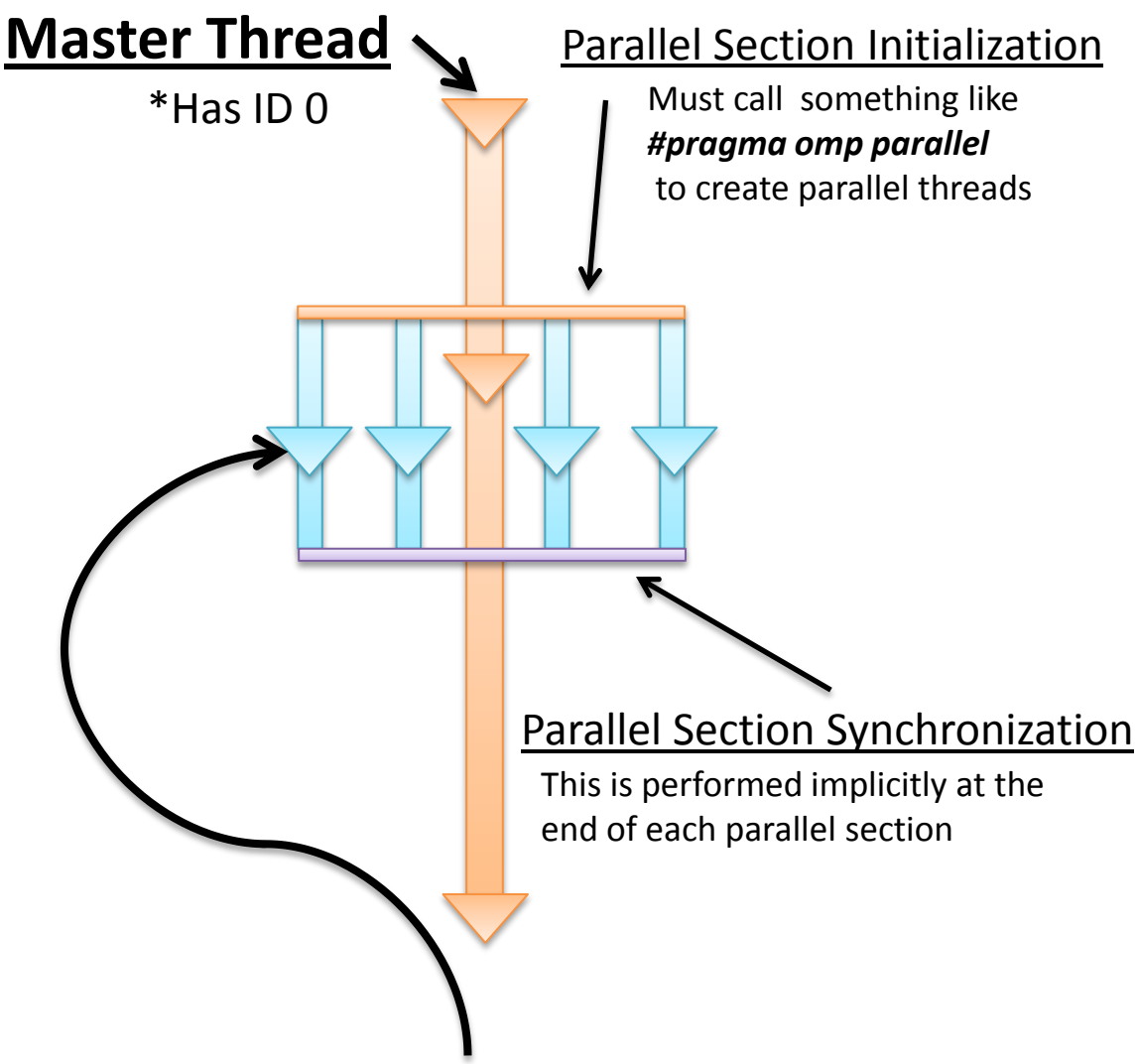
Shared Memory

In OpenMP all the threads can access **Shared Memory Data**. **Private Memory Data** can only be accessed by the thread that owns it.



Fork and Join Model

When a program starts to run, it consists of just one thread. The master thread then uses OpenMP **directives** which initialize executing the **Parallel Section**. When all the threads finish executing their instructions, the results are synchronized and the program can continue through the completion.



helloWorld.c

In a Unix terminal window go to your home directory by typing:

```
cd ~
```

Then edit your .bashrc file. Use your preferred editor, such as vim:

```
vim .bashrc
```

Add the following line, which is an alias to compile OpenMP code:

```
alias ompcc='gcc44 -fopenmp'
```

helloWorld.c

Let's create a Hello World program.

- In your terminal type

```
vim hello.c
```

- Include OpenMP library by typing the following inside of your hello.c file.

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

- Add the following compiler directive inside your main function to create many threads that will display a hello world message:

```
#pragma omp parallel
{
    /*Parallel Section*/
}
```

The code should look like the sample below:

```
#include <stdio.h>
#include <omp.h>
int main(){
    #pragma omp parallel
    {
        printf("Hello World!\n");
    }
    return (0);
}
```

helloWorld.c

Compile and run the code using the following command lines in the Unix terminal window:

Compile: `ompcc helloWorld.c`
Run: `./a.out`

Expected Output:

Multiple hello world messages.

Each message line corresponds to a thread created by OpenMP. Since we did not specify how many threads we wanted, OpenMP creates one thread for each processor available in the run.

helloWorld.c

Instead of counting how many hello world messages were output, the following function can tell how many threads OpenMP created in the run:

```
omp_get_num_threads();
```

Let's see how many processors are available. The code should look similar to this:

```
#include <stdio.h>
#include <omp.h>
  int main(){

    int numThreads;

    #pragma omp parallel
    {
       numThreads = omp_get_num_threads();
       printf("Hello World! There are %d
    threads\n",numThreads);
    }
  return (0);
  }
```

Compile and run the code:

Compile: `ompcc helloWorld.c`

Run: `./a.out`

helloWorld.c

Expected Output:

The number of threads should correspond to the number of hello world messages.

It is possible to specify the number of threads that Open MP will create. To do so use the following function:

```
int omp_set_num_threads(int num);
```

helloWorld.c

In this example 5 threads will be created.

With that modification, the code should correspond to the example below:

```
#include <stdio.h>
#include <omp.h>
  int main(){
    omp_set_num_threads(5);
    int numThreads;

    #pragma omp parallel
    {
       numThreads = omp_get_num_threads();
       printf("Hello World! There are %d
    threads\n",numThreads);
    }
  return (0);
  }
```

Compile and run the code:

Compile: `ompcc helloWorld.c`

Run: `./a.out`

helloWorld.c

Expected Output:
5 hello world messages.

If there are 5 threads in this run, then each thread should have an ID. To display which corresponds to each hello world message, use the following function call to get the individual thread numbers:

```
omp_get_thread_num();
```

helloWorld.c

Let's output the thread number with the hello world message. The code should look like this:

```
#include <stdio.h>
#include <omp.h>
  int main(){
    omp_set_num_threads(5);
    int numThreads, tNum;

    #pragma omp parallel
    {
       numThreads = omp_get_num_threads();
       tNum = omp_get_thread_num();
       printf("Hello World! I am thread %d.
    There are %d threads\n",
    tNum,numThreads);
    }
  return (0);
  }
```

Compile and run the code:

Compile: `ompcc helloWorld.c`

Run: `./a.out`

helloWorld.c

Expected Output:

Each hello world message should contain its own thread ID.

(Note: Thread numbering starts from zero.)

Try running the code a few times. Is there anything unexpected? Are there two or more messages with the same thread ID number?

If the answer is “yes” then congratulations! This is data contention called a ***Race Condition***!

Race Condition: Occurs when multiple threads change the value of a variable at the same time. In this case, the variable can get overwritten before it gets output by the thread. In this case, tNum is the variable getting overwritten.

There are two ways to fix this ***Race Condition***.

The first is to add the compiler directive ***critical*** after the **#pragma omp parallel** statement, but that will make the code execute one thread at a time. Thus, defeating the purpose of parallel programming.

helloWorld.c

The other way of doing this is to add the compiler directive ***private()*** after the **#pragma omp parallel** statement, for every variable added inside private's parenthesis, OpenMP will create an instance of it for each thread. Thus, each thread will have its own copy and won't overwrite anyone's variable.

Your code should look similar to this now:

```
#include <stdio.h>
#include <omp.h>
  int main(){
    omp_set_num_threads(5);
    int numThreads, tNum;

    #pragma omp parallel private(tNum)
    {
       numThreads = omp_get_num_threads();
       tNum = omp_get_thread_num();
       printf("Hello World! I am thread %d.
    There are %d threads\n",
    tNum,numThreads);
    }
  return (0);
  }
```

Compile and Run the program.

helloWorld.c

Expected Output

If everything was done correctly, then the output should be the same as before. This time there is no race condition, regardless of how many times the program is run, because tNum is not getting overwritten, since each thread has its own tNum.

It probably seems excessive to output the number of threads with each hello world message, doesn't it? Then how can the code be modified to display this information only once?

Each thread has its own number. Thus, it is possible to give instructions to a specific thread. By simply creating an if-else statement it is possible to give thread zero, for example, the instructions to print how many threads there are and let the other threads just to display the hello world message.

Why is this important to learn?

In computational Science, scientists are working with large problems. Before the calculations are done, work has to be divided among the processors, to increase the performance one thread is in charge of distributing the work and eventually gathering the results, while the rest of the threads are used to do calculations.

helloWorld.c

The following code will tell thread zero to display how many threads there are and leave the other threads to say hello world.

```
#include <stdio.h>
#include <omp.h>
  int main(){
    omp_set_num_threads(5);
    int numThreads, tNum;

    #pragma omp parallel private(tNum)
    {
       tNum = omp_get_thread_num();
       if(tNum == 0) {
         numThreads = omp_get_num_threads();
         printf("Hello World! I am thread
    %d. There are %d threads.\n",
    tNum,numThreads);
       }
       else {
       printf("Hello World from thread
    %d.\n", tNum);
       }
    }
  return (0);
  }
```

helloWorld.c

Compile and Run the program.

Expected Output

Five hello world messages, with thread zero displaying:

"Hello World! I am thread 0. There are 5 threads."

Parallel Programming

If everything was done correctly then it is time to move to something more advanced. For example, finding the area under a curve in parallel!

Using everything learned in this module, let's try to find the area under the x^2 curve, in the domain X[0, 1].

There is a serial version of code that finds the area under a curve on the next page. Try to parallelize this code using OpenMP.

integration.c

Serial Code:

```
float f(float x) {
    return (x*x);
}
int main()
    int i, SECTIONS = 1000;
    float height = 0.0;
    float area = 0.0, y = 0.0, x = 0.0;
    float dx = 1.0/(float)SECTIONS;

        for( i = 0; i < SECTIONS; i++){
            x = i*dx;
            y = f(x);
            area += y*dx;
        }
 printf("Area under the curve is %f\n",area);

    return (0);
}
```

integration.c

Compile and run the serial version using the same commands as before. The area under the x^2 curve from X[0,1] should be approximately 0.333.

Turn this serial code into a parallel code by simply adding one directive in the line above of the for loop.

```
#pragma omp parallel for private(var)
reduction(operation:var)
```

(note: The directive should be all in one line)

The two new directives on this line are:

`for:` This part of the directive tells the compiler that the next line is a for loop and then it divides the loop into equal parts for each thread, but the order that the loop is performed is nondeterministic, in other words, it does not perform the loop in chronological order.

`reduction:` Creates a private copy of each variable for each thread. After the parallel section has ended, all the copies are combined using the operator specified, such as "**+**"(addition), "**-**"(subtraction), "*****"(multiplication), etc.

integration.c

The code should be similar to the one below:

```
float f(float x) {
    return (x*x);
}
int main()
    int i, PRECISION = 1000;
    float height = 0.0;
    float area = 0.0, y = 0.0, x = 0.0;
    float dx = 1.0/(float)PRECISION;
        #pragma omp parallel for private(x, y)
reduction(+: area)
        for( i = 0; i < PRECISION; i++){
            x = i*dx;
            y = f(x);
            area += y*dx;
        }
  printf("Area under the curve is %f\n",area);

    return (0);
}
```

Expected Output

The area under the curve should be approximately 0.333

More about OpenMP

To learn more about OpenMP, here is a list of website that can be useful use to learn more about OpenMP.

- ✓ http://openmp.org/
- ✓ http://en.wikipedia.org/wiki/OpenMP
- ✓ https://computing.llnl.gov/tutorials/openMP/

These sites include good summaries summary of all OpenMP's directives, functions, environment variables.

- ✓ http://www.openmp.org/mp-documents/OpenMP3.0-SummarySpec.pdf
- ✓ http://www.openmp.org/mp-documents/OpenMP3.0-FortranCard.pdf