Introduction

Because Summit is a cluster of CPUs, the most effective way to utilize these resources involve parallel programming. Probably the simplest way to begin parallel programming involves the utilization of OpenMP. OpenMP is a Compiler-side solution for creating code that runs on multiple cores/threads. Because OpenMP is built into a compiler, no external libraries need to be installed in order to compile this code. These tutorials will provide basic instructions on utilizing OpenMP on both the GNU C++ Compiler and the Intel C++ Compiler.

This guide assumes you have basic knowledge of the command line and the C++ Language.

Resources:

Much Deeper OpenMP and MPI C++ tutorial:
http://pages.tacc.utexas.edu/~eijkhout/pcse/html/index.html

Helpful Slides from the Aristotle University of Thessaloniki:
https://training.vi-seem.eu/images/trainingMaterial/HPC/OpenMP/HP-SEE-SS_2011-OpenMP.pdf

Parallel "Hello, World" Program

In this section we will learn how to make a simple parallel hello world program in C++.
Let's begin with the creation of a program titled: parallel_hello_world.cpp. From the command line run the command:

```
nano parallel_hello_world.cpp
```

We will begin with include statements we want running at the top of the program:

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

These flags allow us to utilize the stdio and omp libraries in our program. The 'omp.h' header file will provide openmp functionality. The 'stdio.h' header file will provide us with print functionality.

Let's now begin our program by constructing the main function of the program. We will use omp_get_thread_num() to obtain the thread id of the process. This will let us identify each of our threads using that unique id number.

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

```
#include <omp.h>

int main(int argc, char** argv){
      printf(“Hello from process: %d\n”, omp_get_thread_num());
      return 0;
}
```

Let’s compile our code and see what happens. We must first load the compiler module we want into our environment. We can do so as such:

GCC

```
module load gcc
```

Or

Intel

```
module load intel
```

From the command line, where your code is located, run the command:

GCC

```
g++ parallel_hello_world.cpp -o parallel_hello_world.exe -fopenmp
```

Or

Intel

```
icc parallel_hello_world.cpp -o parallel_hello_world.exe -qopenmp
```

This will give us an executable we can submit as a job to Summit. Simply submit the job specifying slurm to run the executable. Your submission script should look something like this:

```
#!/bin/bash

#SBATCH --nodes=1
#SBATCH --time=0:01:00
#SBATCH --qos=debug
#SBATCH --partition=shas
#SBATCH --ntasks=4
#SBATCH --job-name=CPP_Hello_World
#SBATCH --output=CPP_Hello_World.out

./parallel_hello_world.exe
```

Our output file should look like this:

```
Hello from process: 0
```

As you may have noticed, we only get one thread giving us a Hello statement.
How do we parallelize the print statement? We parallelize it with **pragma**!

The `#pragma omp parallel { ... }` directive creates a section of code that will be run in parallel by multiple threads. Let’s implement it in our code:

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

int main(int argc, char** argv){

      #pragma omp parallel
      {
            printf(“Hello from process: %d\n”, omp_get_thread_num());
      }
      return 0;
}
```

We must do one more thing before achieving parallelization. To set the amount of threads we want OpenMP to run on, we must set an Linux environment variable to be specify how many threads we wish to use. The environment variable: `OMP_NUM_THREADS` will store this information. On the command line type:

```
export OMP_NUM_THREADS=4
```

Important to note: this environment variable will need to be set every time you exit your shell. If you would like to make this change permanent you will need to add this line your .bash_profile file in your home directory:

```
OMP_NUM_THREADS=4; export OMP_NUM_THREADS
```

Now let’s re-compile the code and run it to see what happens:

Compile with GCC

```
g++ parallel_hello_world.cpp -o parrallel_hello_world.exe -fopenmp
```

Or

Compile with Intel

```
icc parallel_hello_world.cpp -o parrallel_hello_world.exe -qopenmp
```

Resubmit our job script and we should end with an output file similar to this one:

```
Hello from process: 3
Hello from process: 0
Hello from process: 2
Hello from process: 1
```

(Note don’t worry about order of processes that printed, the threads will print out at varying times.)

Private vs. Shared Variables

Memory management is a quintessential component of any parallel program that involves data manipulation. In this section, we will learn about the different variable types in OpenMP as well as a simple implementation of these types into the program we made in the previous section.

OpenMP has a variety of tools that can be utilized to properly describe how the parallel program should handle variables. These tools come in the forms of **shared** and **private** variable type classifiers.

- Private types create a copy of a variable for each process in the parallel system.
- Shared types hold one instance of a variable for all processes to share.

To indicate private or shared memory, declare the variable before your parallel section and annotate the pragma omp directive as such:

```
#pragma omp shared(shar_Var1) private(priv_Var1, priv_Var2)
```

Variables that are created and assigned inside of a parallel section of code will be inherently be private, and variables created outside of parallel sections will be inherently public.

Example:
Let’s adapt our ‘Hello World’ code to utilize private variables as an example.
Starting with the code we left off with, let’s create a variable to store the thread id of each process.

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

```
int main(int argc, char** argv){

    int thread_id;

    #pragma omp parallel
    {
        printf(“Hello from process: %d\n”, omp_get_thread_num());
    }
    return 0;
}
```

Now let’s define `thread_id` as a private variable. Because we want each task to have a unique thread id, using the `private(thread_id)` will create a separate instance of `thread_id` for each task.

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

int main(int argc, char** argv){

    int thread_id;

    #pragma omp parallel private(thread_id)
    {
        printf(“Hello from process: %d\n”, omp_get_thread_num());
    }
}
```

Lastly, let’s assign the thread id to our private variable and print out the variable instead of the `omp_get_thread_num()` function call:

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

int main(int argc, char** argv){

    int thread_id;

    #pragma omp parallel private(thread_id)
    {
        thread_id = omp_get_thread_num();
        printf(“Hello from process: %d\n”, thread_id);
    }
```

```
            return 0;
      }
```

Compiling and submitting our code will result in a similar result to our original hello world:

```
Hello from process: 3
Hello from process: 0
Hello from process: 2
Hello from process: 1
```

Barrier and Critical Directives

OpenMP has a variety of tools for managing processes. One of the more prominent forms of control comes with the **barrier:**

```
      #pragma omp barrier
```

...and the **critical** directives:

```
      #pragma omp critical { … }
```

The barrier directive stops all processes for proceeding to the next line of code until all processes have reached the barrier. This allows a programmer to synchronize sequences in the parallel process.

A critical directive ensures that a line of code is only run by one process at a time, ensuring thread safety in the body of code.

Example:
Let's implement an OpenMP barrier by making our ‘Hello World’ program print its processes in order. Beginning with the code we created in the previous section, let’s nest our print statement in a loop which will iterate from 0 to the max thread count. We will retrieve the max thread count using the OpenMP function:

```
      omp_get_max_threads()
```

Our ‘Hello World’ program will now look like:

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

      int main(int argc, char** argv){
        //define loop iterator variable outside parallel region
        int i;
        int thread_id;

        #pragma omp parallel
        {
            thread_id = omp_get_thread_num();

            //create the loop to have each thread print hello.
            for(i = 0; i < omp_get_max_threads(); i++){
                printf(“Hello from process: %d\n”, thread_id);
            }
        }
        return 0;
    }
```

Now that the loop has been created, let’s create a conditional that requires the loop to be on the proper iteration to print its thread number:

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

    int main(int argc, char** argv){

        int i;
        int thread_id;

        #pragma omp parallel
        {

            for(i = 0; i < omp_get_max_threads(); i++){
                if(i == thread_ID){
                    printf(“Hello from process: %d\n”, thread_id);
                }
            }
        }
        return 0;
    }
```

Lastly, to ensure one process doesn’t get ahead of another, we need to add a barrier directive in the code. Let’s implement one in our loop:

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

```
#include <omp.h>

int main(int argc, char** argv){

    int i;
    int thread_id;

    #pragma omp parallel
    {

        for( int i = 0; i < omp_get_max_threads(); i++){
            if(i == omp_get_thread_num()){
                printf(“Hello from process: %d\n”, thread_id);
            }
            #pragma omp barrier
        }
    }
    return 0;
}
```

Compiling and submitting our code should order our print statements as such:

```
Hello from process: 0
Hello from process: 1
Hello from process: 2
Hello from process: 3
```

Work Sharing Directive: omp for

OpenMP’s power comes from easily splitting a larger task into multiple smaller tasks. Work-sharing directives allow for simple and effective splitting of normally serial tasks into fast parallel sections of code. In this section we will learn how to implement **omp for** directive.

The directive **omp for** divides a normally serial for loop into a parallel task. We can implement this directive as such:

```
#pragma omp for { … }
```

Example:
Let’s write a program to add all the numbers between 1 and 1000. Begin with a main function and the stdio and omp headers:

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

```
#include <omp.h>

int main(int argc, char** argv){
      return 0;
}
```

Now let’s go ahead and setup variables for our parallel code. Lets first create variables partial_Sum and total_Sum to hold each thread’s partial summation and to hold the total sum of all threads respectively.

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

int main(int argc, char** argv){

      int partial_Sum, total_Sum;

      return 0;
}
```

Next let’s begin our parallel section with pragma omp parallel. We will also set partial_Sum to be a private variable and total_Sum to be a shared variable. We shall initialize each variable in the parallel section.

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

int main(int argc, char** argv){

      int partial_Sum, total_Sum;

      #pragma omp parallel private(partial_Sum) shared(total_Sum)
      {
            partial_Sum = 0;
            total_Sum = 0;
      }
      return 0;
}
```

Let’s now set up our work sharing directive. We will use the #pragma omp for to declare the loop as to be work sharing, followed by the actual C++ loop. Because we want to add all number from 1 to 1000, we will initialize out loop at one and end at 1000.

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

int main(int argc, char** argv){

      int partial_Sum, total_Sum;

      #pragma omp parallel private(partial_Sum) shared(total_Sum)
      {
            partial_Sum = 0;
            total_Sum = 0;

            #pragma omp for
            {
                  for(int i = 1; i <= 1000; i++){
                        partial_Sum += i;
                  }
            }
      }
      return 0;
}
```

Now we must join our threads. To do this we must use a critical directive to create a thread safe section of code. We do this with `#pragma omp critical` directive. Lastly we add partial sum to total sum and print out the result outside the parallel section of code.

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

int main(int argc, char** argv){

      int partial_Sum, total_Sum;

      #pragma omp parallel private(partial_Sum) shared(total_Sum)
      {
            partial_Sum = 0;
            total_Sum = 0;

            #pragma omp for
            {
                  for(int i = 1; i <= 1000; i++){
                        partial_Sum += i;
                  }
            }

            //Create thread safe region.
            #pragma omp critical
            {
                  //add each threads partial sum to the total sum
                  total_Sum += partial_Sum;
            }
      }
      printf(“Total Sum: %d\n”, total_Sum);
      return 0;
}
```

This will complete our parallel summation. Compiling and submitting our code will result in this output:

```
Total Sum: 500500
```