Introduction

Because Summit is a cluster of CPUs, parallel programming is the most effective way to utilize these resources. Probably the simplest way to begin parallel programming is 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 provide basic instructions on utilizing OpenMP on both the GNU Fortran Compiler and the Intel Fortran Compiler.

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

Resources:

Much Deeper OpenMP Fortran tutorial:
http://www.openmp.org/wp-content/uploads/F95_OpenMPv1_v2.pdf

Parallel "Hello, World" Program

In this section we will learn how to make a simple parallel hello world program in Fortran. Let's begin with creation of a program titled: parallel_hello_world.f90. From the command line run the command:

```
nano parallel_hello_world.f90
```

We will begin with the program title and the use statement at the top of the program:

```
PROGRAM Parallel_Hello_World
USE OMP_LIB
```

These flags allow us to utilize the omp library in our program. The 'USE OMP_LIB' line of code will provide openmp functionality.

Let's now begin our program by constructing the main body 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.

```
PROGRAM Parallel_Hello_World
USE OMP_LIB

PRINT *, "Hello from process: ", OMP_GET_THREAD_NUM()
```

```
END
```

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:

GNU Fortran
```
module load gcc
```

Or

Intel Fortran
```
module load intel
```

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

GNU Fortran
```
gfortran parallel_hello_world.f90 -o parallel_hello_world.exe -fopenmp
```

Or

Intel Fortran
```
ifort parallel_hello_world.f90 -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=Fortran_Hello_World
#SBATCH --output=Fortran_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 **omp parallel**!

The `!$OMP PARALLEL` and `!$OMP END PARALLEL` directives creates a section of code that is run from all available threads.

```
PROGRAM Parallel_Hello_World
USE OMP_LIB

!$OMP PARALLEL

      PRINT *, “Hello from process: ”, OMP_GET_THREAD_NUM()

!$OMP END PARALLEL

END
```

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.

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

GNU Fortran
```
gfortran parallel_hello_world.f90 -o parallel_hello_world.exe -fopenmp
```

Or

Intel Fortran
```
ifort parallel_hello_world.f90 -o parallel_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 indicate how the parallel program should handle variables. These tools come in the forms of **shared** and **private** variable classifiers.

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

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

```
!$OMP PARALLEL 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. We will also change the name of the program as good coding practice.

```
PROGRAM Parallel_Stored_Hello
USE OMP_LIB

INTEGER :: thread_id

!$OMP PARALLEL

      PRINT *, “Hello from process: ”, OMP_GET_THREAD_NUM()

!$OMP END PARALLEL

END
```

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.

```
PROGRAM Parallel_Stored_Hello
USE OMP_LIB

INTEGER :: thread_id

!$OMP PARALLEL PRIVATE(thread_id)

      PRINT *, “Hello from process: ”, OMP_GET_THREAD_NUM()

!$OMP END PARALLEL

END
```

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:

```
PROGRAM Parallel_Stored_Hello
USE OMP_LIB

INTEGER :: thread_id

!$OMP PARALLEL PRIVATE(thread_id)

      thread_id = OMP_GET_THREAD_NUM()
      PRINT *, “Hello from process: ”, thread_id

!$OMP END PARALLEL

END
```

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:**

```
!$OMP BARRIER
```

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

```
!$OMP CRITICAL
...
!$OMP END 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 processes in the parallel program.

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:

```
PROGRAM Parallel_Ordered_Hello
USE OMP_LIB

INTEGER :: thread_id

!$OMP PARALLEL PRIVATE(thread_id)

      thread_id = OMP_GET_THREAD_NUM()

      DO i=1,OMP_GET_MAX_THREADS()
            PRINT *, "Hello from process: ", thread_id
      END DO

      !$OMP END PARALLEL
```

```
END
```

Now that the loop has been created, let’s create a conditional that will stop a process from printing its thread number until the loop iteration matches its thread number:

```
PROGRAM Parallel_Ordered_Hello
USE OMP_LIB

INTEGER :: thread_id

!$OMP PARALLEL PRIVATE(thread_id)

      thread_id = OMP_GET_THREAD_NUM()

      DO i=1,OMP_GET_MAX_THREADS()
            IF (i == thread_id) THEN
                  PRINT *, "Hello from process: ", thread_id
            END IF
      END DO

!$OMP END PARALLEL

END
```

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.

```
PROGRAM Parallel_Ordered_Hello
USE OMP_LIB

INTEGER :: thread_id

!$OMP PARALLEL PRIVATE(thread_id)

      thread_id = OMP_GET_THREAD_NUM()

      DO i=1,OMP_GET_MAX_THREADS()
            IF (i == thread_id) THEN
                  PRINT *, "Hello from process: ", thread_id
            END IF
            !$OMP BARRIER
      END DO
!$OMP END PARALLEL

END
```

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 do

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 do** directive.

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

```
      !$OMP DO
      ...
      !$OMP END DO
```

Example:
Let's write a program to add all the numbers between 1 and 1000. Begin with a program title and the omp_lib header:

```
      PROGRAM Parallel_Do
      USE OMP_LIB

      END
```

Now let's go ahead and setup variables for our parallel code. Let's 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.

```
      PROGRAM Parallel_Hello_World
      USE OMP_LIB

      INTEGER :: partial_Sum, total_Sum

      END
```

Next let's begin our parallel section with `!$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.

```
PROGRAM Parallel_Hello_World
USE OMP_LIB

INTEGER :: partial_Sum, total_Sum

!$OMP PARALLEL PRIVATE(partial_Sum) SHARED(total_Sum)

      partial_Sum = 0;
      total_Sum = 0;

!$OMP END PARALLEL

END
```

Let's now set up our work sharing directive. We will use the `!$OMP DO` to declare the loop to be work sharing, followed by the actual Fortran loop. Because we want to add all number from 1 to 1000, we will initialize out loop at one and end at 1000.

```
PROGRAM Parallel_Hello_World
USE OMP_LIB

INTEGER :: partial_Sum, total_Sum

!$OMP PARALLEL PRIVATE(partial_Sum) SHARED(total_Sum)

      partial_Sum = 0;
      total_Sum = 0;

      !$OMP DO
      DO i=1,1000
            partial_Sum = partial_Sum + i
      END DO

!$OMP END PARALLEL

END
```

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 the `!$OMP CRITICAL` directive. Lastly we add partial sum to total sum and print out the result outside the parallel section of code.

```
PROGRAM Parallel_Hello_World
USE OMP_LIB

INTEGER :: partial_Sum, total_Sum

!$OMP PARALLEL PRIVATE(partial_Sum) SHARED(total_Sum)

      partial_Sum = 0;
      total_Sum = 0;

      !$OMP DO
      DO i=1,1000
            partial_Sum = partial_Sum + i
      END DO
      !$OMP END DO

      !$OMP CRITICAL
            total_Sum = total_Sum + partial_Sum
      !$OMP END CRITICAL

!$OMP END PARALLEL
PRINT *, “Total Sum: ”, total_Sum

END
```

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

```
Total Sum: 500500
```