Introduction

Parallel programs enable users to fully utilize the multi-node structure of supercomputing clusters. **Message Passing Interface (MPI)** is a standard used to allow different nodes on a cluster to communicate with each other. In this tutorial we will be using the Intel Fortran Compiler, GCC, IntelMPI, and OpenMPI to create a multiprocessor programs in Fortran.

This tutorial assumes the user has experience in both the Linux terminal and Fortran.

Resources

Helpful MPI tutorials:
http://www.dartmouth.edu/~rc/classes/intro_mpi/intro_mpi_overview.html
http://condor.cc.ku.edu/~grobe/docs/intro-MPI.shtml
https://computing.llnl.gov/tutorials/mpi/

Setup and "Hello World"

Begin by logging into the cluster and using ssh to log in to a Summit compile node. This can be done with the command:

```
ssh scompile
```

Next we must load MPI into our environment. Begin by loading in the Fortran compiler and OpenMPI. Use the following commands if using the GNU Fortran compiler:

GNU Fortran Compiler

```
module load gcc
module load openmpi
```

Or, use the following commands if you prefer to use the Intel Fortran compiler:

Intel Fortran Compiler

```
module load intel
module load impi
```

This should prepare your environment with all the necessary tools to compile and run your MPI code. Let's now begin to construct our Fortran program. In this tutorial, we will name our program file: `hello_world_mpi.f90`

Open `hello_world_mpi.f90` and begin by including the mpi library `'mpi.h'`, and titling the program `hello_world_mpi`

```
PROGRAM hello_world_mpi

include  'mpif.h'
```

Now let's set up several MPI directives to parallelize our code. In this 'Hello World' tutorial we will be calling the following four functions from the MPI library:

MPI_INIT(): This function initializes the MPI environment. It takes in the an error handling variable.
MPI_COMM_SIZE(): This function returns the total size of the environment in terms of the quantity of processes. The function takes in the MPI environment, an integer to hold the commsize, and an error handling variable.
MPI_COMM_RANK(): This function returns the process id of the process that called the function. The function takes in the MPI environment, an integer to hold the comm rank, and an error handling variable.
MPI_FINALIZE(): This function cleans up the MPI environment and ends MPI communications.

These four directives are enough to get our parallel 'hello world' program running. We will begin by creating three integer variables, `process_Rank`, `size_Of_Cluster`, and `ierror` to store an identifier for each of the parallel processes, store the number of processes running in the cluster, and handle error codes respectively. We will also implement the `MPI_Init` function which will initialize the mpi communicator:

```
PROGRAM hello_world_mpi

include  'mpif.h'

integer process_Rank, size_Of_Cluster, ierror

call MPI_INIT(ierror)
```

Let's now obtain some information about our cluster of processors and print the information out for the user. We will use the functions `MPI_Comm_size()` and `MPI_Comm_rank()` to obtain the count of processes and the rank of a given process respectively:

```
PROGRAM hello_world_mpi

include  'mpif.h'

integer process_Rank, size_Of_Cluster, ierror

call MPI_INIT(ierror)
```

```
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)

print *, 'Hello World from process: ', rank, 'of ', size
```

Lastly let's close the environment using `MPI_Finalize()`:

```
PROGRAM hello_world_mpi

include  'mpif.h'

integer rank, size, ierror, tag
call MPI_INIT(ierror)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)

print *, 'Hello World from process: ', rank, 'of ', size

call MPI_FINALIZE(ierror)

END PROGRAM
```

Now the code is complete and ready to be compiled. Because this is an MPI program, we have to use a specialized compiler. The compilation command will be one of the following:

OpenMPI

```
mpif90 hello_world_mpi.f90 -o hello_world_mpi.exe
```

Intel MPI

```
mpiifort hello_world_mpi.f90 -o hello_world_mpi.exe
```

This will produce an executable we can submit to Summit as a job.

In order to execute MPI compiled code, a special command must be used:

```
mpirun -np 4 ./hello_world_mpi.exe
```

The flag `-np` specifies the number of processor that are to be utilized in execution of the program.

In your job submission script, load the same compiler and openMPI choices you used above to create and compile the program, and submit the job with slurm to run the executable. Your job submission script should look something like this:

OpenMPI

```
#!/bin/bash

#SBATCH -N 1
#SBATCH --ntasks 4
#SBATCH --job-name parallel_hello
#SBATCH --partition shas
#SBATCH --time 0:01:00
#SBATCH --output parallel_hello_world.out
#SBATCH --qos Debug

module purge
module load gcc
module load openmpi

mpirun -np 4 ./hello_world_mpi.exe
```

Intel MPI

```
#!/bin/bash

#SBATCH -N 1
#SBATCH --ntasks 4
#SBATCH --job-name parallel_hello
#SBATCH --partition shas
#SBATCH --time 0:01:00
#SBATCH --output parallel_hello_world.out
#SBATCH --qos Debug

module purge
module load intel
module load impi

mpirun -np 4 ./hello_world_mpi.exe
```

It is important to note that on Summit, there are 24 cores per node. For applications that require more than 24 processes, you will need to request multiple nodes in your job submission (i.e., "-N <number of nodes>").

Our output file should look something like this (note the order of ranks isn't necessarily sequential):

```
Hello World from process 3 of 4
Hello World from process 2 of 4
Hello World from process 1 of 4
Hello World from process 0 of 4
```

Ref: http://www.dartmouth.edu/~rc/classes/intro_mpi/hello_world_ex.html

MPI Barriers and Synchronization

Like many other parallel programming utilities, synchronization is an essential tool in thread safety and ensuring certain sections of code are handled at certain points. `MPI_BARRIER` is a process lock that holds each process at a certain line of code until all processes have reached that line. `MPI_BARRIER` can be called as such:

```
MPI_BARRIER(MPI_com comm, integer ierror);
```

To get a handle on barriers, let's modify our "Hello World" program so that it prints out each process in order of thread id. Starting with our "Hello World" code from the previous section, begin by putting our print statement in a loop:

```
PROGRAM hello_world_mpi

include  'mpif.h'

integer rank, size, ierror

call MPI_INIT(ierror)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)

DO i = 0, 3, 1
      print *, 'Hello World from process: ', rank, 'of ', size
END DO

call MPI_FINALIZE(ierror)
```

```
END PROGRAM
```

Next, let's implement a conditional statement in the loop to print only when the loop iteration matches the process rank.

```
PROGRAM hello_world_mpi

include  'mpif.h'

integer rank, size, ierror

call MPI_INIT(ierror)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)

DO i = 0, 3, 1
      IF(i == rank) THEN
            print *, 'Hello World from process: ', rank, 'of ', size
      END IF
END DO

call MPI_FINALIZE(ierror)

END PROGRAM
```

Lastly, implement the barrier function in the loop. This will ensure that all processes are synchronized when passing through the loop.

```
PROGRAM hello_world_mpi

include  'mpif.h'

integer rank, size, ierror

call MPI_INIT(ierror)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)

DO i = 0, 3, 1
      IF(i == rank) THEN
            print *, 'Hello World from process: ', rank, 'of ', size
```

```
        END IF
        call MPI_BARRIER( MPI_COMM_WORLD, i_error)
    END DO

    call MPI_FINALIZE(ierror)

    END PROGRAM
```

Compiling and submitting this code will result in the following output (note the ranks are now sequential):

```
Hello World from process 0 of 4
Hello World from process 1 of 4
Hello World from process 2 of 4
Hello World from process 3 of 4
```

Message Passing

Message passing is the primary utility in the MPI application interface that allows for processes to communicate with each other. Next, we will learn the basics of message passing between two processes.

Message passing in MPI is handled by the corresponding functions and their arguments:

```
MPI_SEND(integer message, integer count, MPI_Datatype datatype, integer dest,
      integer tag, MPI_Comm comm, integer ierror);

MPI_RECV(integer data, integer count, MPI_Datatype datatype, integer from,
      integer tag, MPI_Comm comm, MPI_Status* status, integer ierror);
```

The arguments are as follows:

MPI_SEND

integer message:	Variable storing message you are sending.
integer count:	Number of elements being sent through the array.
MPI_Datatype datatype:	The MPI-specific data type being passed through the array.
integer dest:	Process rank of destination process.
integer tag:	Message tag.
MPI_Comm comm:	The MPI Communicator handle.
integer ierror:	An error handling variable.

MPI_RECV

integer message:	Variable storing message you are receiving.
integer count:	Number of elements being sent through the array.
MPI_Datatype datatype:	The MP-specific data type being passed through the array.
integer from:	Process rank of sending process.
integer tag:	Message tag.
MPI_Comm comm:	The MPI Communicator handle.
MPI_Status* status:	Status object.
integer ierror:	An error handling variable.

Let's implement message passing in an example:

Example

We will pass the number 42 from one process to another. We will use our "Hello World" program as a starting point for this program. Let's begin by renaming our program and creating a variable to store some information.

```
PROGRAM send_recv_mpi

include  'mpif.h'

integer rank, size, ierror, message_Item

call MPI_INIT(ierror)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)

call MPI_FINALIZE(ierror)

END PROGRAM
```

Now create 'if' and 'else if' conditionals that specify the appropriate processes to call MPI_SEND() and MPI_RECV() functions. In this example we want process 1 to send out a message containing the integer 42 to process 2.

```
PROGRAM send_recv_mpi

include  'mpif.h'

integer rank, size, ierror, message_Item

call MPI_INIT(ierror)
```

```
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)

IF(rank == 0) THEN
      message_Item = 42
      print *, "Sending message containing: ", message_Item
ELSE IF(rank == 1) THEN
      print *, "Received message containing: ", message_Item
END IF

call MPI_FINALIZE(ierror)

END PROGRAM
```

Lastly we must call MPI_SEND() and MPI_RECV(). We will pass in the following parameters into the functions:

```
MPI_SEND(
      message_Item,          //Variable storing the message we are sending.
      1,                     //Number of elements handled by the array.
      MPI_INT,               //MPI_TYPE of the message we are sending.
      1,                     //Rank of receiving process
      1,                     //Message Tag
      MPI_COMM_WORLD         //MPI Communicator
      ierror                 //Error Handling Variable
)

MPI_RECV(
      message_Item,          //Variable storing the message we are receiving.
      1,                     //Number of elements handled by the array.
      MPI_INT,               //MPI_TYPE of the message we are sending.
      0,                     //Rank of sending process
      1,                     //Message Tag
      MPI_COMM_WORLD         //MPI Communicator
      MPI_STATUS_IGNORE      //MPI Status Object
      ierror                 //Error Handling Variable
)
```

Lets implement these functions in our code:

```
PROGRAM send_recv_mpi

include  'mpif.h'
```

```
integer rank, size, ierror, message_Item

call MPI_INIT(ierror)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)

IF(rank == 0) THEN
      message_Item = 42
      call MPI_SEND(message_Item, 1, MPI_INT, 1, 1, MPI_COMM_WORLD,
            ierror)
      print *, "Sending message containing: ", message_Item
ELSE IF(rank == 1) THEN
      call MPI_RECV(message_Item, 1, MPI_INT, 0, 1, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE, ierror)
      print *, "Received message containing: ", message_Item
END IF

call MPI_FINALIZE(ierror)

END PROGRAM
```

Compiling and submitting a batch job with our code that requests 2 processes (--ntasks 2) will result in the following output:

```
Sending message containing: 42
Received message containing: 42
```

Group Operators: Scatter and Gather

Group operators are very useful for MPI. They allow for swaths of data to be distributed from a root process to all other available processes, or data from all processes can be collected at one process. These operators can eliminate the need for a surprising amount of boilerplate code via two functions:

MPI_Scatter

integer send_Var:	Variable storing the values that will be scattered.
integer send_Count:	Number of elements that will be scattered.
MPI_Datatype send_Type:	MPI Datatype of the data that is scattered.
integer recv_Var:	Variable that will store the scattered data.
integer recv_Count:	Number of data elements that will be received per process.
MPI_Datatype recv_Type:	MPI Datatype of the data that will be received.

integer root_Process:	The rank of the process that will scatter the information.
MPI_Comm comm:	The MPI_Communicator.
integer ierror:	An error handling variable.

MPI_Gather

integer send_Var:	Variable storing the value that will be sent.
integer send_Count:	Number of data elements that will sent .
MPI_Datatype send_Type:	MPI Datatype of the data that is sent.
integer recv_Var:	Variable that will store the gathered data.
integer recv_Count:	Number of data elements per process that will be received.
MPI_Datatype recv_Type:	MPI Datatype of the data that will be received.
integer root_Process:	The rank of the process rank that will gather the information.
MPI_Comm comm:	The MPI_Communicator.
integer ierror:	An error handling variable.

In order to get a better grasp on these functions, let's go ahead and create a program that will utilize the scatter function. Note that the gather function (not shown in the example) works similarly, and is essentially the converse of the scatter function. Further examples which utilize the gather function can be found in the MPI tutorials listed as resources at the beginning of this document.

Example

We will create a new program that scatters one element of a data array to each process. Specifically, this code will scatter the four elements of a vector array to four different processes. We will start with a Fortran header along with variables to store process rank and number of processes.

```
PROGRAM scatter_mpi

include 'mpif.h'
integer rank, size, ierror, message_Item

END PROGRAM
```

Now let's setup the MPI environment using `MPI_Init`, `MPI_Comm_size`, `MPI_Comm_rank`, and `MPI_Finaize`:

```
PROGRAM scatter_mpi

include 'mpif.h'
```

```
integer rank, size, ierror, message_Item

call MPI_INIT(ierror)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)

call MPI_FINALIZE(ierror)

END PROGRAM
```

Next let's generate an array named `distro_Array` to store four numbers. We will also create a variable called `scattered_Data` to which we will scatter the data.

```
PROGRAM scatter_mpi

include 'mpif.h'
integer rank, size, ierror, message_Item
integer scattered_Data

integer, dimension(4) :: distro_Array
distro_Array = (/39, 72, 129, 42/)

call MPI_INIT(ierror)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)

call MPI_FINALIZE(ierror)

END PROGRAM
```

Now we will begin the use of group operators. We will use the operator scatter to distribute distro_Array into scattered_Data. Let's take a look at the parameters we will use in this function:

```
MPI_Scatter(
    distro_Array,           //Array we are scattering from.
    1,                      //Number of items we are sending each processor
    MPI_INT,                //MPI Datatype of scattering array.
    scattered_Data,         //Variable to which are receiving scattered data.
    1,                      //Amount of data each process will receive.
    MPI_INT,                //MPI Datatype of receiver array.
    0,                      //Process ID that will distribute the data.
    MPI_COMM_WORLD          //MPI Communicator.
    ierror                  //Error Handling Variable
```

)

Let's implement this in the code. We will also write a print statement following the scatter call:

```
PROGRAM scatter_mpi

include 'mpif.h'
integer rank, size, ierror, message_Item
integer scattered_Data

integer, dimension(4) :: distro_Array
distro_Array = (/39, 72, 129, 42/)

call MPI_INIT(ierror)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)

call MPI_Scatter(distro_Array, 1, MPI_INT, scattered_Data, 1, MPI_INT,
0, MPI_COMM_WORLD, ierror);

print *, "Process ", rank, "received: ", scattered_Data

call MPI_FINALIZE(ierror)

END PROGRAM
```

Running this code will print out the four numbers in the distro array as four separate numbers each from different processes (note the order of ranks isn't necessarily sequential):

```
Process 1 received: 39
Process 0 received: 72
Process 3 received: 129
Process 2 received: 42
```