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 several different processors on a cluster to communicate with each other. In this tutorial we will be using the Intel C++ Compiler, GCC, IntelMPI, and OpenMPI to create a multiprocessor 'hello world' program in C++.

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

Resources

Helpful MPI tutorials:
http://www.dartmouth.edu/~rc/classes/intro_mpi/intro_mpi_overview.html
http://mpitutorial.com/tutorials/
http://condor.cc.ku.edu/~grobe/docs/intro-MPI-C.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 compile node. This can be done with the command:

```
ssh scompile
```

Next we must load MPI into our environment. Begin by loading in your choice of C++ compiler and its corresponding MPI library. Use the following commands if using the GNU C++ compiler:

GNU C++ Compiler
```
module load gcc
module load openmpi
```

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

Intel C++ 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 C++ file. In this tutorial, we will name our code file: `hello_world_mpi.cpp`

Open `hello_world_mpi.cpp` and begin by including the C standard library `<stdio.h>` and the MPI library `<mpi.h>`, and by constructing the main function of the C++ code:

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

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

Now let's set up several MPI directives to parallelize our code. In this 'Hello World' tutorial we'll be utilizing the following four directives:

MPI_Init(): This function initializes the MPI environment. It takes in the addresses of the C++ command line arguments argc and argv.
MPI_Comm_size(): This function returns the total size of the environment via quantity of processes. The function takes in the MPI environment, and the memory address of an integer variable.
MPI_Comm_rank(): This function returns the process id of the processor that called the function. The function takes in the MPI environment, and the memory address of an integer variable.
MPI_Finalize(): This function cleans up the MPI environment and ends MPI communications.

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

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

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

	MPI_Init(&argc, &argv);

	return 0;
}
```

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 process respectively:

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

int main(int argc, char** argv){
	int process_Rank, size_Of_Cluster;

	MPI_Init(&argc, &argv);
	MPI_Comm_size(MPI_COMM_WORLD, &size_Of_Cluster);
	MPI_Comm_rank(MPI_COMM_WORLD, &process_Rank);

	printf("Hello World from process %d of %d\n", process_Rank,
		size_Of_Cluster);

	return 0;
}
```

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

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

int main(int argc, char** argv){
	int process_Rank, size_Of_Cluster;

	MPI_Init(&argc, &argv);
	MPI_Comm_size(MPI_COMM_WORLD, &size_Of_Cluster);
	MPI_Comm_rank(MPI_COMM_WORLD, &process_Rank);

	printf("Hello World from process %d of %d\n", process_rank,
		size_Of_Cluster);

	MPI_Finalize();

	return 0;
}
```

Now the code is complete and ready to be compiled. Because this is an MPI program, we have to use a specialized compiler. Be sure to use the correct command based off of what compiler you have loaded.

OpenMPI

```
mpic++ hello_world_mpi.cpp -o hello_world_mpi.exe
```

Intel MPI

```
mpiicc hello_world_mpi.cpp -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 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 is a total of 24 cores per node. For applications that require more than 24 processes, you will need to request multiple nodes in your job submission.

Our output file should look something like this:

```
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 in code. `MPI_Barrier` can be called as such:

```
MPI_Barrier(MPI_Comm comm);
```

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 nesting our print statement in a loop:

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

int main(int argc, char** argv){
      int process_Rank, size_Of_Cluster;
```

```
	MPI_Init(&argc, &argv);
	MPI_Comm_size(MPI_COMM_WORLD, &size_Of_Cluster);
	MPI_Comm_rank(MPI_COMM_WORLD, &process_Rank);

	for(int i = 0, i < size_Of_Cluster, i++){
		printf("Hello World from process %d of %d\n", process_Rank,
			size_Of_Cluster);
	}

	MPI_Finalize();

	return 0;
}
```

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

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

int main(int argc, char** argv){
	int process_Rank, size_Of_Cluster;

	MPI_Init(&argc, &argv);
	MPI_Comm_size(MPI_COMM_WORLD, &size_Of_Cluster);
	MPI_Comm_rank(MPI_COMM_WORLD, &process_Rank);

	for(int i = 0, i < size_Of_Cluster, i++){
		if(i == process_Rank){
			printf("Hello World from process %d of %d\n",
				process_Rank, size_Of_Cluster);
		}
	}

	MPI_Finalize();

	return 0;
}
```

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

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

int main(int argc, char** argv){
	int process_Rank, size_Of_Cluster;

	MPI_Init(&argc, &argv);
	MPI_Comm_size(MPI_COMM_WORLD, &size_Of_Cluster);
	MPI_Comm_rank(MPI_COMM_WORLD, &process_Rank);

	for(int i = 0, i < size_Of_Cluster, i++){
		if(i == process_Rank){
			printf("Hello World from process %d of %d\n",
				process_Rank, size_Of_Cluster);
		}
		MPI_Barrier(MPI_COMM_WORLD);
	}

	MPI_Finalize();

	return 0;
}
```

Compiling and submitting this code will result in this output:

```
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. In this tutorial, we will learn the basics of message passing between 2 processes.

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

```
MPI_Send(void* message, int count, MPI_Datatype datatype, int dest,
		int tag, MPI_Comm, communicator);

MPI_Recv(void* data, int count, MPI_Datatype datatype, int from,
		int tag, MPI_Comm comm, MPI_Status* status);
```

The arguments are as follows:.

MPI_Send

void* message:	Address for the message you are sending.
int count:	Number of elements being sent through the address.
MPI_Datatype datatype:	The MPI specific data type being passed through the address.
int dest:	Rank of destination process.
int tag:	Message tag.
MPI_Comm comm:	The MPI Communicator handle.

MPI_Recv

void* message:	Address to the message you are receiving.
int count:	Number of elements being sent through the address.
MPI_Datatype datatype:	The MPI specific data type being passed through the address.
int from:	Process rank of sending process.
int tag:	Message tag.
MPI_Comm comm:	The MPI Communicator handle.
MPI_Status* status:	Status object.

Let's implement message passing in an example:

Example

We will create a two-process process that 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 creating a variable to store some information.

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

int main(int argc, char** argv){
      int process_Rank, size_Of_Cluster, message_Item;

      MPI_Init(&argc, &argv);
      MPI_Comm_size(MPI_COMM_WORLD, &size_Of_Cluster);
      MPI_Comm_rank(MPI_COMM_WORLD, &process_Rank);

      MPI_Finalize();

      return 0;
```

```
}
```

Now create ‘if’ and ‘else if’ conditionals that specify appropriate process 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.

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

int main(int argc, char** argv){
	int process_Rank, size_Of_Cluster, message_Item;

	MPI_Init(&argc, &argv);
	MPI_Comm_size(MPI_COMM_WORLD, &size_Of_Cluster);
	MPI_Comm_rank(MPI_COMM_WORLD, &process_Rank);

	if(process_Rank == 0){
		message_Item = 42;
		printf(“Sending message containing: %d\n”, message_Item)
	}
	else if(process_Rank == 1){
		printf(“Received message containing: %d\n”, message_Item)
	}

	MPI_Finalize();

	return 0;
}
```

Lastly we must call MPI_Send() and MPI_Recv(). We will pass the following parameters into the functions:

```
MPI_Send(
	&message_Item,		//Address of the message we are sending.
	1,			//Number of elements handled by that address.
	MPI_INT,		//MPI_TYPE of the message we are sending.
	1,			//Rank of receiving process
	1,			//Message Tag
	MPI_COMM_WORLD		//MPI Communicator
)

MPI_Recv(
	&message_Item,		//Address of the message we are receiving.
```

```
        1,                      //Number of elements handled by that address.
        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
)
```

Lets implement these functions in our code:

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

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

      int process_Rank, size_Of_Cluster, message_Item;

      MPI_Init(&argc, &argv);

      MPI_Comm_size(MPI_COMM_WORLD, &size_Of_Cluster);
      MPI_Comm_rank(MPI_COMM_WORLD, &process_Rank);

      if(process_Rank == 0){
            message_Item = 42;
            MPI_Send(&message_Item, 1, MPI_INT, 1, 1, MPI_COMM_WORLD);
            printf("Message Sent: %d\n", message_Item);
      }
      else if(process_Rank == 1){
            MPI_Recv(&message_Item, 1, MPI_INT, 0, 1, MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);
            printf("Message Received: %d\n", message_Item);
      }

      MPI_Finalize();

      return 0;
}
```

Compiling and submitting our code with 2 processes will result in the following output:

```
Message Sent: 42
Message Received: 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 the use of two functions:

MPI_Scatter

void* send_Var:	Address of the variable that will be scattered.
int send_Count:	Number of elements that will be scattered.
MPI_Datatype send_Type:	MPI Datatype of the data that is scattered.
void* recv_Var:	Address of the variable that will store the scattered data.
int 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.
int root_Process:	The rank of the process that will scatter the information.
MPI_Comm comm:	The MPI_Communicator.

MPI_Gather

void* send_Var:	Address of the variable that will be sent.
int send_Count:	Number of data elements that will sent .
MPI_Datatype send_Type:	MPI Datatype of the data that is sent.
void* recv_Var:	Address of the variable that will store the received data.
int 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.
int root_Process:	The rank of the process rank that will gather the information.
MPI_Comm comm:	The MPI_Communicator.

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 program that scatters one element of a data array to each process. Specifically, this code will scatter the four elements of an array to four different processes. We will start with a basic C++ main function along with variables to store process rank and number of processes.

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

```
int main(int argc, char** argv){
	int process_Rank, size_Of_Comm;

	return 0;
}
```

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

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

int main(int argc, char** argv){
	int process_Rank, size_Of_Comm;

	MPI_Init(&argc, &argv);

	MPI_Comm_size(MPI_COMM_WORLD, &size_Of_Comm);
	MPI_Comm_rank(MPI_COMM_WORLD, &process_Rank);

	MPI_Finalize();

	return 0;
}
```

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

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

int main(int argc, char** argv){
	int process_Rank, size_Of_Comm;
	int distro_Array[4] = {39, 72, 129, 42};
	int scattered_Data;

	MPI_Init(&argc, &argv);

	MPI_Comm_size(MPI_COMM_WORLD, &size_Of_Comm);
	MPI_Comm_rank(MPI_COMM_WORLD, &process_Rank);

	MPI_Finalize();
```

```
	return 0;
}
```

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,		Address of array we are scattering from.
	1,			//Number of items we are sending each processor
	MPI_INT,		MPI Datatype of scattering array.
	&scattered_Data,	Address of array we 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.
)
```

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

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

int main(int argc, char** argv){
	int process_Rank, size_Of_Comm;
	int distro_Array[4] = {39, 72, 129, 42};
	int scattered_Data;

	MPI_Init(&argc, &argv);

	MPI_Comm_size(MPI_COMM_WORLD, &size_Of_Comm);
	MPI_Comm_rank(MPI_COMM_WORLD, &process_Rank);

	MPI_Scatter(&distro_Array, 1, MPI_INT, &scattered_Data, 1,
			MPI_INT, 0, MPI_COMM_WORLD);

	printf("Process has received: %d \n", scattered_Data);

	MPI_Finalize();

	return 0;
}
```

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

```
Process has received: 39
Process has received: 72
Process has received: 129
Process has received: 42
```