L14 – Supercomputing - Part 2

1. MPI Code Structure

Writing parallel code can be done in either C or Fortran. The **Message Passing Interface** (**MPI**) is just a set of subroutines that has bindings in either language. That is, we write our codes as normal and use the MPI subroutine set to handle the details of communication between processors for us. We just need to worry about when that communication takes place and what is said.

MPI has many subroutines (125 total functions), however it is really easy to work with and many programs can be written using just 6 functions.

The six main functions of MPI are:

1) **MPI_Init** – Initialize MPI environment
2) **MPI_Finalize** – Finalize MPI environment
3) **MPI_Comm_size** – Determine total number of processors
4) **MPI_Comm_rank** – Determine the rank of the current processor
5) **MPI_Send** – Send a message
6) **MPI_Recv** – Receive a message

All MPI programs will have the same basic structure. The main elements are organized as follows:

```
PROGRAM example_mpi
USE mpi
IMPLICIT NONE
INTEGER :: mpi_ierr, nprocs, mpi_rank

CALL MPI_Init(mpi_ierr)

CALL MPI_Comm_size(MPI_COMM_WORLD, nprocs, mpi_ierr)

CALL MPI_Comm_rank(MPI_COMM_WORLD, mpi_rank, mpi_ierr)

!  Do your calculations here, i.e., the main program elements

CALL MPI_Finalize(mpi_ierr)


END PROGRAM example_mpi
```

Note that what we do is:

- We start out by initializing the MPI environment. This is done with just the **MPI_Init** subroutine. All this does is say start up MPI. We have designated the variable **mpi_ierr** to tell us about the status of each MPI action we perform.

- Next we find out how many processors we are actually using. Here we use the subroutine **MPI_Comm_size** and place the result into the variable **nprocs**.

- Next we find out which processing rank we are. We use the subroutine **MPI_Comm_rank** and place the result into the variable **mpi_rank**.

- Now the MPI environment is completely set up and we can write the main part of the code.

- Once everything is done we need to close off the MPI environment with the subroutine **MPI_Finalize**.

2. Your First Parallel Code

Often times when we start to write code our first code is a **Hello** program. We will do this with MPI because it is a little more interesting than in the normal case. Our examples will be shown for the environment of the University of Utah's Center for High Performance Computing (CHPC). Most of you will have an account on one of CHPC's computers (if not join up with someone who does for this exercise) and log in now to sanddunearch:

```
>> ssh –X –l username sanddunearch.chpc.utah.edu
```

Our basic hello world program is **mpiexample.f90**:

```
PROGRAM example
USE mpi
IMPLICIT NONE
INTEGER :: mpi_ierr, nprocs, mpi_rank

! initialize MPI environment
CALL MPI_Init(mpi_ierr)
CALL MPI_Comm_size(MPI_COMM_WORLD, nprocs, mpi_ierr)
CALL MPI_Comm_rank(MPI_COMM_WORLD, mpi_rank, mpi_ierr)

! just have rank 0 state how many processors we are using
IF (mpi_rank == 0) THEN
  write(*,*) nprocs, "processes have been requested."
ENDIF

! Here is the hello world part...
write(*,*) "Hi, I am Rank: ", mpi_rank

CALL MPI_Finalize(mpi_ierr)

END PROGRAM example
```

Compiling MPI Codes

The first thing we may now note is that we can't compile MPI code with just the standard g95 type of call. Instead we need to use a Fortran 90 compiler built for MPI. Usually this is called

mpif90. On sanddunearch we have several flavors (similar to all the different flavors of f90). You can see the list of all the compilers CHPC supports on their web page

http://www.chpc.utah.edu/docs/manuals/user_guides/arches/

Here we will use the **pathscale mpif90** compiler since it is my favorite. Hence, to compile we need just to know the path to this compiler:

```
>>/uufs/sanddunearch.arches/sys/pkg/mvapich/std/bin/mpif90
mpiexample.f90 –o mpiexample.x
```

Executing MPI Codes

Note that now you should have an executable file called **mpiexample.x** but that we can **NOT** just type **`./mpiexample.x`** to execute this code.

To execute this code there are two basic ways: (1) through interactive mode, or (2) through the batch system. Typically we will execute our codes through the batch system.

To enter the interactive mode you would type:

```
>> qsub -I -l nodes=1:ppn=4,walltime=10:00
```

Before we go on, note that we are requesting to use **1 node** and **4 processors** (**ppn** = **p**rocessor **p**er **n**ode) for a total time of **10:00 minutes**.

But it is preferable to just create a script that we can submit through the batch system.

To run this code we need to use the program **mpirun**. But, we also need to use the same version of **mpirun** that was set up for the version of **mpif90** that we used above:

```
>> /uufs/sanddunearch.arches/sys/pkg/mvapich/std/bin/mpirun_rsh –
rsh –np 4 –hostfile $PBS_NODEFILE ./mpiexample.x
```

So, to run this job lets create a file: **run_mpi.pbs**

```
#!/bin/bash

#PBS –N testjob
#PBS –A tj-sda
#PBS –l qos=thorne
#PBS –l walltime=00:10:00
#PBS –o test.out
#PBS –e test.err
#PBS –l nodes=4

cd $PBS_O_WORKDIR

/uufs/sanddunearch.arches/sys/pkg/mvapich/std/bin/mpirun_rsh –rsh
–np 4 –hostfile $PBS_NODEFILE ./mpiexample.x
```

We can now submit the job to be run by typing:

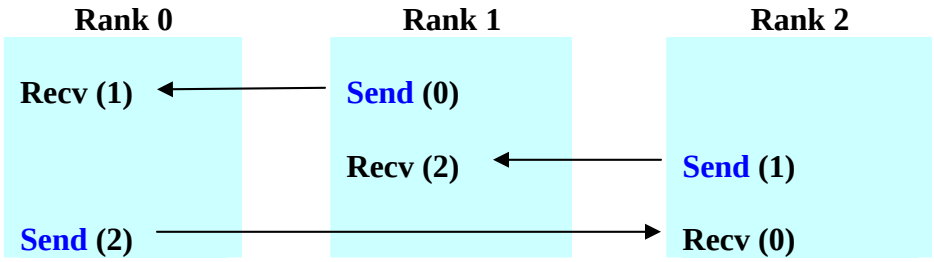
```
>> qsub run_mpi.pbs
```

Note that the output we wrote to screen is written to the file **test.out**. You can see what the program output is by examining this file. This is an example of how to use the batch system to run jobs. We will discuss this further in Section 5, but note that there are directives in this script: **tj-sda** and **qos=thorne** which specifically state to run on my personal nodes. This is fine in the confines of this class, but please don't use my nodes for your personal jobs.

This is of course a really simple example, but note that we got a response from each processor. Now let's take a look at how to pass information between processors.

3. Basic Communication Routines (Send/Receive)

Let's think of a simple example where we want to send a real number to the rank immediately to the right (or wrap around if we are at the farthest right processor). The situation may look like:



The first thing we may want to do is define a variable we will call **torank** that defines which processor rank we want to send information to:

```
IF (myrank > 0) torank = myrank - 1
IF (myrank == 0) torank = 2
```

To actually send this information we use the **MPI_Send** subroutine. The basic format of this subroutine looks like:

```
MPI_Send (buf, count, datatype, dest, tag, comm, ierror)

Where,
buf         = the actual variable to send.
count       = the number of elements to receive
datatype    = the type of data to send
dest        = the rank of the process to send the message to
tag         = an integer number identifying the message
comm        = the communicator (e.g., MPI_COMM_WORLD)
ierror      = the fortran return code.
```

Hence, if the real number we wanted to send was in the variable data we would do:

```
CALL MPI_Send(data,1,MPI_REAL,torank,tag,MPI_COMM_WORLD,mpi_ierr)
```

So far we have told which processor where to send its data to. But, we haven't specified which processors should be listening for data. To receive the data being sent we need to add an **MPI_Recv** call. For our above example:

```
IF (myrank < 2) fromrank = myrank + 1
IF (myrank == 2) fromrank = 0
CALL MPI_Recv(rec,1,MPI_REAL,fromrank,tag,MPI_COMM_WORLD, &
mpi_status,mpi_ierr)
```

The **MPI_Recv** subroutine is quite similar to the **MPI_Send** subroutine:

```
MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror)

Where,
buf         = the actual variable to send.
count       = the number of elements to receive
datatype    = the type of data to send
source      = the rank of the process to receive the message from
tag         = an integer number identifying the message
comm        = the communicator (e.g., MPI_COMM_WORLD)
status      = message status
ierror      = the fortran return code.
```

Let's put this altogether into a program:

```
PROGRAM mpisendexample
USE mpi
IMPLICIT NONE
REAL    :: X, Y
INTEGER :: mpi_ierr, nprocs, myrank, mpi_status
INTEGER :: torank, fromrank, tag


! Initialize the MPI environment
!--------------------------------------------------------------------!
CALL MPI_Init(mpi_ierr)
CALL MPI_Comm_size(MPI_COMM_WORLD,nprocs,mpi_ierr)
CALL MPI_Comm_rank(MPI_COMM_WORLD,myrank,mpi_ierr)
!--------------------------------------------------------------------!


! Let's Make the variable X be something specific to each processor:
!--------------------------------------------------------------------!
X = 10.0*float(myrank)
!--------------------------------------------------------------------!

!--------------------------------------------------------------------!
! Now let's read the value of X from the rank to the right
! and store it in the variable Y
!--------------------------------------------------------------------!
```

```
!First let's send the data
!---------------------------------------------------------------------!
IF (myrank > 0) torank = myrank - 1
IF (myrank == 0) torank = nprocs - 1
tag = 1

write(*,*) "myrank: ", myrank, "sending to rank: ", torank

CALL MPI_Send(X,1,MPI_REAL,torank,tag,MPI_COMM_WORLD,mpi_ierr)
!---------------------------------------------------------------------!

!We will add a barrier here, not because its necessary but so that our
! output comes in a more reasonable fashion for this example
!---------------------------------------------------------------------!
CALL MPI_Barrier(MPI_COMM_WORLD,mpi_ierr)
IF (myrank == 0) write(*,*) "-------------------------"
!---------------------------------------------------------------------!

!Now Let's receive it!
!---------------------------------------------------------------------!
IF (myrank < (nprocs-1)) fromrank = myrank + 1
IF (myrank == (nprocs-1)) fromrank = 0

write(*,*) "myrank: ", myrank, "receiving from rank: ", fromrank

CALL
MPI_Recv(Y,1,MPI_REAL,fromrank,tag,MPI_COMM_WORLD,mpi_status,mpi_ierr)
!---------------------------------------------------------------------!

!We will add a barrier here as well
!---------------------------------------------------------------------!
CALL MPI_Barrier(MPI_COMM_WORLD,mpi_ierr)
!---------------------------------------------------------------------!

! Now let's report what do we have
!---------------------------------------------------------------------!
IF (myrank == 0) write(*,*) "-------------------------"
write(*,*) "On rank: '", myrank, "'; X =", X, " and Y = ", Y
!---------------------------------------------------------------------!

CALL MPI_Finalize(mpi_ierr)

END PROGRAM mpisendexample
```

It is useful to note here that in our send and receive messages we had to specify that we were sending a real number with MPI_REAL. The primary data types you will use in Fortran are:

MPI_REAL
MPI_INTEGER
MPI_CHARACTER

4. Some useful MPI functions

The above example utilized another function called MPI_Barrier. The action of the Barrier function is to synchronize processes. That is it essentially halts the program until all of the processors have reached the Barrier call. We used it above so that the output would be written in a little more sequential manner. Nonetheless, it wasn't necessary.

As noted there are over 100 MPI functions. You can find what they are and their syntax at the following web page:

http://www.dei.unipd.it/~addetto/manuali_online/SP/MPISubRef/d3d80mst02.html

But, let's review a couple of the most useful here so you can see how these functions work in general.

MPI_BCAST – Which is short for **B**road**CAST**. With the broadcast command one processor sends the same message to a number of recipients with a single operation.

Let's look at a simple example that reads in some information to **rank 0** and then **broadcasts** that information to all other processors.

Let's first create a file **input.txt** with some information to read in (a character, an integer, and a real number):

```
my_input_example
10
13.567
```

Our code might look like:

```
PROGRAM mpisendexample
USE mpi
IMPLICIT NONE
REAL    :: realnum
INTEGER :: nr
INTEGER :: mpi_ierr, nprocs, myrank, mpi_status
CHARACTER(LEN=30) :: title

! Initialize the MPI environment
!-----------------------------------------------------------------!
CALL MPI_Init(mpi_ierr)
CALL MPI_Comm_size(MPI_COMM_WORLD,nprocs,mpi_ierr)
CALL MPI_Comm_rank(MPI_COMM_WORLD,myrank,mpi_ierr)
!-----------------------------------------------------------------!

! Read in file from standard input on rank 0
!-----------------------------------------------------------------!
IF (myrank == 0) THEN
  read(*,*) title
  read(*,*) nr
  read(*,*) realnum
ENDIF
!-----------------------------------------------------------------!
```

```
! Now let's send this information to all of the other processors
!--------------------------------------------------------------------!
CALL MPI_BCAST(title,   30, MPI_CHARACTER, 0, MPI_COMM_WORLD, mpi_ierr)
CALL MPI_BCAST(realnum, 1,  MPI_REAL,      0, MPI_COMM_WORLD, mpi_ierr)
CALL MPI_BCAST(nr,      1,  MPI_INTEGER,   0, MPI_COMM_WORLD, mpi_ierr)
!--------------------------------------------------------------------!

! Now let's write out the value of one of these variables on the
!   other processors to check that it worked
!--------------------------------------------------------------------!
write(*,*) "On rank: ", myrank, "; title= ", title
!--------------------------------------------------------------------!

CALL MPI_Finalize(mpi_ierr)

END PROGRAM mpisendexample
```

Note that to run this code we would direct the input into the code through standard in:

```
>> mpirun_rsh –rsh –np 4 –hostfile $PBS_NODEFILE ./mpiexample.x < input.txt
```

MPI_allreduce – This handy little utility lets you choose a variable and find the minimum or maximum value of the variable across all ranks and put the output of the action in another variable. Here's an example:

```
PROGRAM mpiredex
USE mpi
IMPLICIT NONE
REAL    :: X, MinX, MaxX
INTEGER :: mpi_ierr, nprocs, myrank, mpi_status

! Initialize the MPI environment
!--------------------------------------------------------------------!
CALL MPI_Init(mpi_ierr)
CALL MPI_Comm_size(MPI_COMM_WORLD,nprocs,mpi_ierr)
CALL MPI_Comm_rank(MPI_COMM_WORLD,myrank,mpi_ierr)
!--------------------------------------------------------------------!
! Make some dummy variable X
!--------------------------------------------------------------------!
X = 0.5*float(myrank+2)
write(*,*) "Rank: ", myrank, "; X= ", X
!--------------------------------------------------------------------!
! Find the min and max of X across all ranks and store in the
!  variables MinX and MaxX
!--------------------------------------------------------------------!
CALL MPI_AllReduce(X,MinX,1,MPI_REAL,MPI_MIN,MPI_COMM_WORLD,mpi_ierr)
CALL MPI_AllReduce(X,MaxX,1,MPI_REAL,MPI_MAX,MPI_COMM_WORLD,mpi_ierr)
write(*,*) "Rank: ", myrank, "; MinX= ", MinX
write(*,*) "Rank: ", myrank, "; MaxX= ", MaxX
!--------------------------------------------------------------------!

CALL MPI_Finalize(mpi_ierr)
END PROGRAM mpiredex
```

Obviously there are many more MPI functions we could talk about. But, to be honest, many of the parallel codes I've written haven't needed to use any other functions than the ones I covered in this lecture.

5. The Batch System

The final thing we need to talk about is submitting jobs. On the CHPC computers here at UU we use the **P**ortable **B**atch **S**ystem (**PBS**) for job scheduling. Other supercomputers may use other systems but they all basically work in the same way although there might be slight differences in syntax. So, as noted a typical batch script may look as follows:

```
#!/bin/bash

#PBS -N testjob
#PBS -A tj-sda
#PBS -l qos=thorne
#PBS -l walltime=00:10:00
#PBS -o test.out
#PBS -e test.err
#PBS -l nodes=4
#PBS -M michael.thorne@utah.edu
#PBS -m ab

cd $PBS_O_WORKDIR

/uufs/sanddunearch.arches/sys/pkg/mvapich/std/bin/mpirun_rsh -rsh
-np 4 -hostfile $PBS_NODEFILE ./mpiexample.x
```

You can find a description of the flags at:

http://www.chpc.utah.edu/docs/manuals/user_guides/arches/#batch

The most important points are:

- We must specify an amount of time (the **walltime**) that the job will require. If your job exceeds the **walltime** it will get **killed**!
- You must specify how many processors to use. On **sanddunearch** this is just done with the **nodes** option.
- When executing the code, you must **again** specify how many processors to use. This is done with the flag **–np**.

To submit the code we use the **qsub** command:

```
>> qsub pbs_script
```

Once we have submitted the code, we can check its status by just typing **showq** or **qstat**. However, I find this to be a little annoying since it shows everyone's jobs.

Hence, it is useful to create an alias that might just show your jobs. For example,

```
alias q = "qstat –a | grep username"
```

Where you will obviously substitute in your own username.

6. Homework

This is another buy week as not all students have access to the supercomputing facilities.