Part 2 – Programming using MPI

Addisu G. Semie, PhD
Asst. Prof.,Computational Science Program,
Addis Ababa University
Email: addisugezahegn@gmail.com

Introduction to MPI

- In this section we will cover:
  - Introduction to MPI
  - Steps in MPI implementation
  - Installing OpenMPI
  - Testing OpenMPI

Introduction to MPI

- MPI (Message Passing Interface) is a message-passing library interface specification

- MPI refers primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process.

- It is a language-independent communications protocol used to program parallel computers.

Introduction to MPI

- The specification is based on the consensus of the MPI Forum, which has over 40 participating organizations, including vendors, researchers, software library developers, and users.
- MPI is the first standardized, vendor independent, message passing library.
- MPI closely match the design goals of portability, efficiency, and flexibility
- It become the "industry standard" for writing message passing programs on HPC platforms.

Introduction to MPI

- The first standard for Message passing Interface (MPI-1) is released in 1994 and followed with subsequent revisions.
- Refer the following link:
  https://www.open-mpi.org/software/ompi/versions/timeline.php

Introduction to MPI

- The message passing model demonstrates the following characteristics:
  - A set of tasks that uses their own local memory during computation.
  - Multiple tasks can reside on the same physical machine as well as across an arbitrary number of machines.
  - Tasks exchange data through communications by sending and receiving messages.
  - Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.

Reasons for Using MPI

- Standardization - MPI is the only message passing library that can be considered as a standard. It is supported on virtually by all HPC platforms.
- Portability - There is little or no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.
- Performance Opportunities - Vendor implementations should be able to exploit native hardware features to optimize performance. Any implementation is free to develop optimized algorithms.
- Functionality - There are over 430 routines defined in MPI-3, which includes the majority of those in MPI-2 and MPI-1.
- Availability - A variety of implementations are available, both vendor and public domain.

MPI Implementations

- From a programming perspective, message passing implementations commonly comprise a library of subroutines that are embedded in the source code.
- The programmer is responsible for determining all parallelism.
- A large number of message passing libraries are available since the 1980s.
- MPI program can be **Single program** or **Multiple program**
- Message Passing usually thought of in the context of distributed memory, parallel computers
- However, the same code can be run well on
  - A shared memory parallel computer
  - A network of workstations
  - A collection of heterogeneous (different architecture) processors
  - Even on a single workstation

Steps in MPI Implementation

- In manual approach of implementing parallel program, there are steps to follow which fairly guide the programmer to successfully accomplish the job.

- However, there is no single best rule (method) to do manual parallelization.

Step 1: Understand the Problem and the Program

- Undoubtedly, the first step in developing parallel software is to understand the problem that you wish to solve in parallel.

- If you are starting with a serial program, this necessitates to understand the implementation logic of the existing code (program).

Step 2: Make sure that the problem can be parallelized

- Before spending time in an attempt to develop a parallel solution for a problem, determine whether or not the problem is one that can actually be parallelized.

- There are problems which can not be parallelized at all.

- There are also existing implementation which can not be parallelized unless the computation logic (algorithm) is changed.

Example of Parallelizable Problem: (Approximating PI)

- Calculate the approximate value of PI by randomly generating a point in the X-Y plane where value of X and Y are bounded to a set of positive number less than or equals to 1.
  - Perform an experiment and see if it is within the sector of the unit circle or not.
  - Count the number of times it occurs within the unit circle (**Na**) and the total number of times the experiment conducted (N).
  - Divide **Na** by **4*N** which is the approximated value of PI.
  - The ratio is equals to PI when N approaches to infinity.
  - This problem is able to be solved in parallel.
  - Each processor will perform the experiment independently for some time and we will gather their findings and find out the estimated value of PI.

Example of a Non-parallelizable Problem:

- Calculation of the Fibonacci series (1,1,2,3,5,8,13,21,...) by use of the formula: **F(k + 2) = F(k + 1) + F(k)**.
  - This is a non-parallelizable problem because the calculation of the Fibonacci sequence as shown above, entail dependent calculations rather than independent ones.
  - The calculation of the k + 2 value uses those of both k + 1 and k.
  - These three terms cannot be calculated independently and therefore, not in parallelizable.

Step 3: Design the parallel Solution

- In order to write programmer directed parallel program we need to identify the program's ***hotspots*** location, the ***bottle necks***, ***inhibitors***, and the ***algorithm*** used.

  Hot Spots

  - Knowing the hot spots includes identifying where most of the real work is being done
  - The majority of scientific and technical programs usually accomplish most of their work in a few places).
  - Profilers and performance analysis tools can help in identifying the hot spots if we have the equivalent serial codes.
  - Once identified we need to focus on parallelizing the hotspots and ignore those sections of the program that account for little CPU usage.

Step 3: Design the parallel Solution

Bottlenecks

- are the areas that are disproportionately slow, or cause parallelizable work to halt or be deferred. (for example, I/O is usually something that slows a program down).
- it may be possible to restructure the program or use a different algorithm to reduce or eliminate unnecessary slow areas.

Inhibitors refers to issues that affect parallelization.

- Some inhibitors include data dependency as demonstrated by the Fibonacci sequence above.
- Finally we should design the parallel algorithm (data parallel or instruction parallel or both).

Step 4: Implement the parallel Algorithm

- The next step is to implement the parallel solution using the selected parallel programming model

Step 5: Debug and test the Program

- This is the time that we need to sit and observe logical flaws that will lead into a wrong logical conclusion.
- We should carefully see all the logic implemented are correct as per the requirement stated in the algorithm design

Step 6: Submit the job to the parallel program manager

- This enable you to run your program in the HPC environment.
- We should be able to know how to submit parallel jobs into the HPC resources.

Step 7: Analyze the result

- This enables you to know how successful you are

Installing OpenMPI

- **OpenMPI** is Open source implementation of MPI standards for use in cluster environment
- Goto the OpenMPI download site (https://www.open-mpi.org/software/ompi/v4.1/)
- Download the stable version for your machine, say openmpi-4.0.5.tar.bz2
- Tar the file
  - tar –xzvf openmpi-4.0.5.tar.bz2

- Goto the folder **openmpi-1.3.2**
  - cd openmpi-4.0.5

Installing OpenMPI

- if you have access to /usr/local (root previlage) you don’t need to specify the option --prefix
  - ./configure
  - make
  - make install

- Otherwise
  make installation directory if you don’t have access to /usr/local
  - ./configure –prefix ~/openmpi/
  - make
  - make install

Installing OpenMPI

- Modify the environment variable **PATH** and **LD_LIBRARY_PATH**
- For CSHELL users open the file ~/.cshrc and add the following into the file
    - setenv PATH {$PATH}:BinPath
    - setenv LD_LIBRARY_PATH {$ LD_LIBRARY_PATH }:LibPath
- Where BinPath is the path for the bin folder of the MPI (example **/usr/local/bin**)
- LibPath is the path for the lib folder of the MPI (example /usr/local/lib)
- For Bash shell open the file ~/.bash_profile and add the following into the file
    - export PATH =$PATH}:BinPath
    - export LD_LIBRARY_PATH ={$ LD_LIBRARY_PATH }:LibPath user
- Close the file and execute the command
  - source ~/.cshrc source ~/.bash_profile

Installing OpenMPI

- To test the program
- Write the following program in your favorite editor and save as ***sample.f90***

- Program sample

```
Include ‘mpif.h’
Call MPI_init(ierror)
Write(*,*) ‘Hello world’
Call MPI_Finalize(ierror)
End program sample
```

Installing OpenMPI

- Compile the program as
  - **mpif90 sample.90 –o sample.x**

- Running the program with n number of processors
  - **mpirun –np 4 sample.x**
  - **mpirun --use-hwthread-cpus -np 8 ./sample.x**
  - **mpirun --oversubscribe -np 16 ./sample.x**

MPI programming basics

- In this section we will cover:
  - General concept and types of MPI function
  - Structure of MPI program
  - Compiling and submitting parallel program

General Concept about MPI functions

- MPI function can be local or non local
- **Local** MPI functions are function whose completion of execution of code doesn’t depend on execution of MPI call by any other process
- **Non-local** MPI function is a function whose completion of execution of code depend on execution of MPI call by another process

General Concept about MPI functions

Terminologies

- ***MPI function call started:*** is to mean the MPI function get CPU access to start execution
- ***MPI function operation started:*** is the time where the other end process start to operate on what this MPI function call need to perform.
- ***MPI function call returns:*** is to mean a called function has returned back to the caller function (it doesn’t mean any thing about the operation)
- ***MPI function operation completed:*** is a condition in which the operation assigned to the MPI function is completed

- Understanding this terms become easier when we think of the function call is from one process but the operation is in another process.

Types of MPI function

- MPI functions are broadly classified into two:
  - Blocking and
  - Non-Blocking
- Blocking
  - The program will not continue until the communication is completed.
- Non-Blocking
  - The program will continue without waiting for communication to be completed.

Structure of MPI program

- All MPI programs have the following general structure:

  include MPI header file

  variable declarations

  initialize the MPI environment

  ...do computation and MPI communication calls...

  close MPI communications

Structure of MPI program

C Programming

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char* argv[]) {
        MPI_Init(&argc, &argv);
        printf("Hello, world\n");
        MPI_Finalize();
        return 0;
}
```

include MPI header file

initialize the MPI environment

...do computation and MPI communication calls...

close MPI communications

End of the program

Fortran Programming

```
Program HelloWorld
        include 'mpi.h'
        Integer :: ierror
        Call MPI_INIT(ierror)
        printf *,  'Hello, world'
        call MPI_FINALIZE(ierror)
end
```

Compiling parallel program in a workstation/PC

- Make sure that you have the compiler installed
  - **mpicc** C Program
  - **mpif77** FORTRAN Program
  - **mpif90** FORTRAN Program
  - **mpic++** C++program
- You may require to specify options as required

Compiling parallel program in a clustered environment

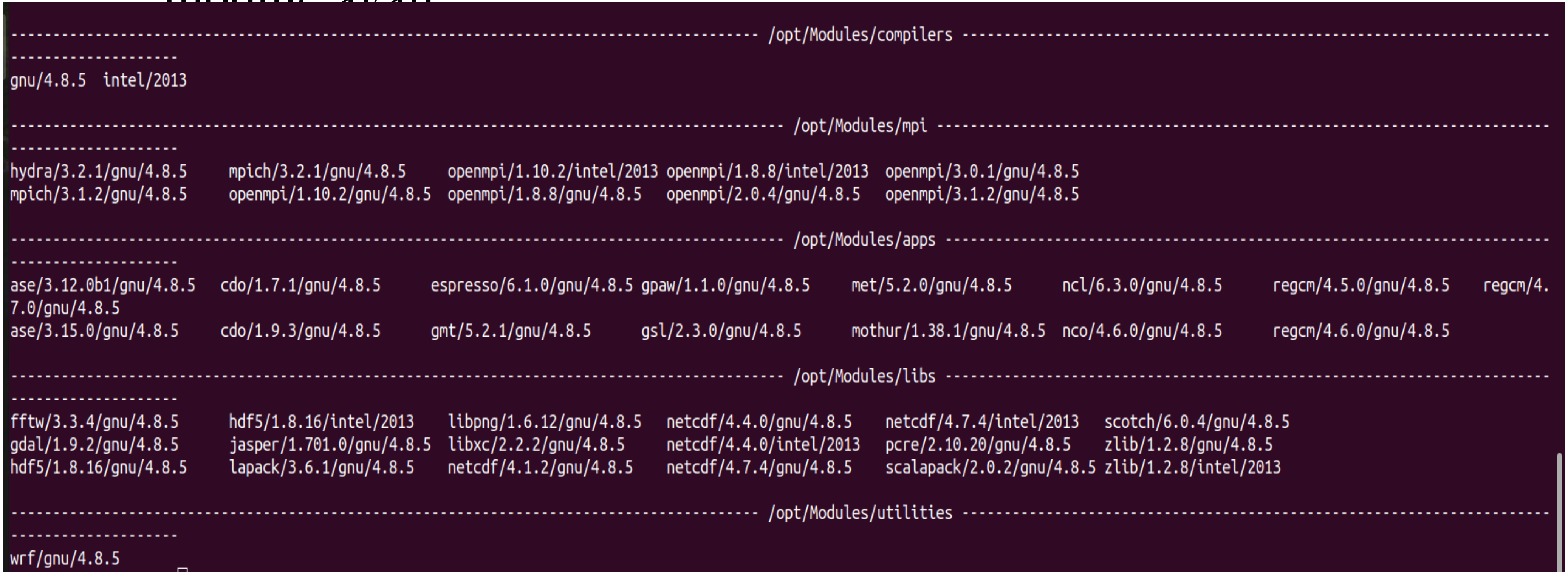
- **Step 1:**
  - Get into the master node of your cluster.
  - This phase usually require you to be authenticated.
  - The logic for the authentication may vary depending on the firewall or the remote server
  - ssh -v addisug@10.4.38.28 !*ssh is a remote login program*
  - It will ask for password and take you into the master node of the cluster (after verifying the authentication)

Compiling parallel program in a clustered environment

- Step 2:
  - Load the required module that will compile your MPI program.
  - Module is a program that enable us to manage resources.
  - The resources are mainly compilers, debuggers, libraries, mpis, and applications.
  - Module program enable us to see, load, unload these resources as required.

Compiling parallel program in a clustered environment

- To see the available modules
  - module avail



```
---------------------------------------------------------------------------------------- /opt/Modules/compilers ---------------------------------------------------------------------------------------
--------------------
gnu/4.8.5  intel/2013

------------------------------------------------------------------------------------------- /opt/Modules/mpi ------------------------------------------------------------------------------------------
--------------------
hydra/3.2.1/gnu/4.8.5     mpich/3.2.1/gnu/4.8.5     openmpi/1.10.2/intel/2013 openmpi/1.8.8/intel/2013  openmpi/3.0.1/gnu/4.8.5
mpich/3.1.2/gnu/4.8.5     openmpi/1.10.2/gnu/4.8.5  openmpi/1.8.8/gnu/4.8.5   openmpi/2.0.4/gnu/4.8.5   openmpi/3.1.2/gnu/4.8.5

------------------------------------------------------------------------------------------- /opt/Modules/apps -----------------------------------------------------------------------------------------
--------------------
ase/3.12.0b1/gnu/4.8.5   cdo/1.7.1/gnu/4.8.5      espresso/6.1.0/gnu/4.8.5 gpaw/1.1.0/gnu/4.8.5     met/5.2.0/gnu/4.8.5      ncl/6.3.0/gnu/4.8.5      regcm/4.5.0/gnu/4.8.5    regcm/4.
7.0/gnu/4.8.5
ase/3.15.0/gnu/4.8.5     cdo/1.9.3/gnu/4.8.5      gmt/5.2.1/gnu/4.8.5      gsl/2.3.0/gnu/4.8.5      mothur/1.38.1/gnu/4.8.5  nco/4.6.0/gnu/4.8.5      regcm/4.6.0/gnu/4.8.5

------------------------------------------------------------------------------------------- /opt/Modules/libs -----------------------------------------------------------------------------------------
--------------------
fftw/3.3.4/gnu/4.8.5      hdf5/1.8.16/intel/2013    libpng/1.6.12/gnu/4.8.5   netcdf/4.4.0/gnu/4.8.5    netcdf/4.7.4/intel/2013   scotch/6.0.4/gnu/4.8.5
gdal/1.9.2/gnu/4.8.5      jasper/1.701.0/gnu/4.8.5  libxc/2.2.2/gnu/4.8.5     netcdf/4.4.0/intel/2013   pcre/2.10.20/gnu/4.8.5    zlib/1.2.8/gnu/4.8.5
hdf5/1.8.16/gnu/4.8.5     lapack/3.6.1/gnu/4.8.5    netcdf/4.1.2/gnu/4.8.5    netcdf/4.7.4/gnu/4.8.5    scalapack/2.0.2/gnu/4.8.5 zlib/1.2.8/intel/2013

---------------------------------------------------------------------------------------- /opt/Modules/utilities ---------------------------------------------------------------------------------------
--------------------
wrf/gnu/4.8.5
```

Compiling parallel program in a clustered environment

- You have to have the required module loaded before you try to compile and execute your job
- To load the desired module
  - module load moduleName
  - module load openmpi/1.3.3/intel/11.1
- To unload a loaded module if it is not required
  - module unload moduleName
  - module unload openmpi/1.3.3/intel/11.1
- To list the set of modules loaded
  - module list
- To clear loaded modules
  - module purge

Compiling parallel program in a clustered environment

- **Step 3:**
  - Compile the program
  - Once you loaded the appropriate compiler, you can compile the program as
  - compilername mpiProgramName -o binaryOutputName
    - mpicc send_receive_and_status.c -o test.x
    - mpif90 send_receive_and_status.f90 -o test.x

- Depending on the complexity and dependencies of modules of the program, you may require to develop MAKEFILE

Submitting parallel program to a workstation

- This approach is used for compiling and debugging and should not be used for production runs as all the created processes are running in a single CPU which doesn’t improve performance.
- To execute your parallel program on a single node (usually on desktops, workstations, or SMP machines), you can use the command
  - mpirun [Options] theProgram
- Some important options
  - **-np <number>**
    - used to indicate the number of processes (processors) needs to be allocated to execute the job
  - **--prefix <path>**
    - this option may be required to locate the libraries where the MPI is installed which is **/usr/local** by default

Submitting parallel program to a Cluster

- This approach is most appropriate to execute our MPI program on a cluster of nodes where each nodes have one or more processors.
- Submission of jobs to one or more nodes require basic knowledge on submission procedure.
- There are two approach on job submission:
  - interactive and
  - batch mode

Submitting parallel program to a Cluster

Interactive job submission procedure

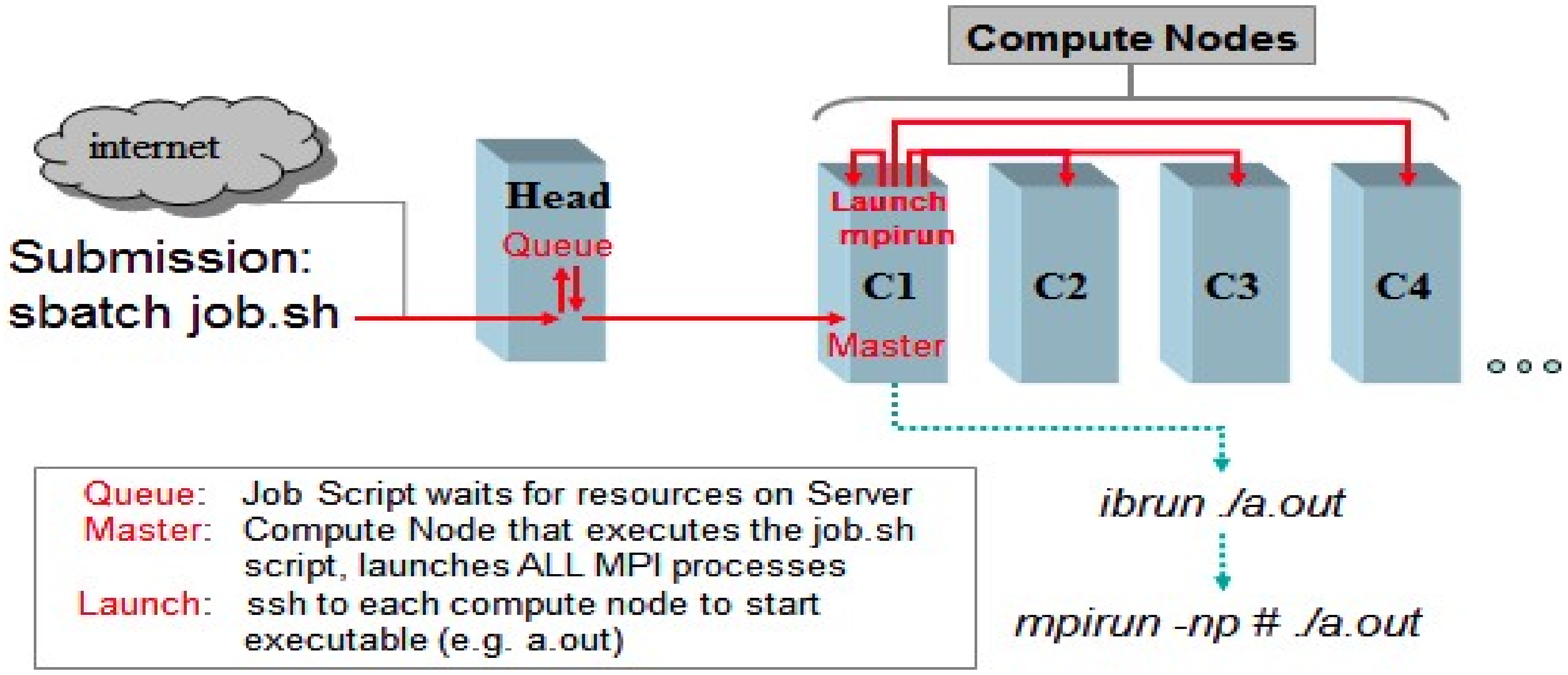
- Interactive mode approach require you to specify the set of commands in sequence (steps) from job submission request till job execution.

- **Step 1:** identify to which cluster of nodes (if you have more than one) you need to submit the job
  - Showbfq !***will help you which has more free slots***
- **Step 2:** request job submission on the command line using qsub command
  - qsub -q parallel -l walltime=3:00:00,nodes=2:ppn=4 -I
  - qsub -q debug -l walltime=5,nodes=2:ppn=4 -I
- On successful request, you will be directed to the home directory of one of the allocated node.
- You may need to change your current directory to the directory where your program is located.

Submitting parallel program to a Cluster

- **Interactive job submission procedure**
- **Step 3:** load the required module to run the program
  - module load openmpi/1.3/intel/11.0
- **Step 4:** Submit the job using the mpirun command
  - mpirun -np 8 test.x
- Here -np specifies the requested number of processes and test.x is the program to be executed
- You can also use more options such as
  - **-npernode <number>** used to indicate how many processor from each node needs to be allocated.
  - If omitted, the task manager will allocate all processors from the first few node (machine) as much as possible.

Submitting parallel program to a Cluster



Queue:	Job Script waits for resources on Server
Master:	Compute Node that executes the job.sh script, launches ALL MPI processes
Launch:	ssh to each compute node to start executable (e.g. a.out)

Submitting parallel program to a Cluster

- **Batch mode submission**
  - To submit job into one or more nodes (worker nodes other than master node), we need to prepare Portable Batch System (PBS) which is a batch job description file.
  - The PBS script is given to the clustered computer system resource management package.
  - The resource manager will accept batch jobs (shell scripts with control attributes), preserve and protect the job until it is run, run the job, and deliver output back to the submitter.
  - The job is specified as a shell script which can contain job management instructions (note that these instructions can also be given in the command line).

Submitting parallel program to a Cluster

- **Batch mode submission**
  - The PBS script file will have two sections:
    - **PBS directives** and
    - **Shell commands** to be executed.
  - The directives are parameters to consider while the PBS execute the batch job where as
  - The commands are the action to be executed in their logical order by the specified shell in each node when execution starts.

Submitting parallel program to a Cluster

- **Batch mode submission**:
  - ***Some useful PBS directives to consider while writing PBS script:***
    - *#PBS -q <queuename>* shows the queue name to which my job should be queued (use qstat -Q to see the possible queue name)
    - *#PBS -e <filename>* *(stderr location, where to place error message)*
      #PBS -o <filename> *(stdout location, where to put output information)*
      #PBS -t <seconds> (maximum time)
      #PBS -l <attribute>=<value>
    - *<attribute>=<value>* can have different forms such as

      nodes=node01 or **nodes=4:ppn=8** or

      nodes=node02:ppn=8 or **nodes=nodes=node01+node05** or

      nodes=node01:ppn=2+node05:ppn=4
    - Assuming that node0j is a node name in the specified queue system.
    - Use the command

      pbsnodes –l all to see all the nodes

Submitting parallel program to a Cluster

- **Batch mode submission**:
  - ***Some use full commands to consider while writing PBS script***
    - module load <ModuleName>
    - module load openmpi/1.3/intel/11.0
  - You may need to provide the name of the required module that needs to be loaded when the job get submitted.
    - cd SPBS_O_WORKDIR
    - This enable the processors to change their directories into the directory where the work is submitted and any relative file location starts from this point

Submitting parallel program to a Cluster

- **Batch mode submission**:
  - ***Some use full commands to consider while writing PBS script***
    - module load <ModuleName>
    - module load openmpi/1.3/intel/11.0
  - You may need to provide the name of the required module that needs to be loaded when the job get submitted.
    - cd SPBS_O_WORKDIR
    - This enable the processors to change their directories into the directory where the work is submitted and any relative file location starts from this point

Submitting parallel program to a Cluster

- **Batch mode submission**:
  - ***Some use full commands to consider while writing PBS script***
    - mpirun [options] parallelProgram
  - Some important options includes
    - **-np <number>**
    - used to indicate the number of processes (processors) needs to be allocated to execute the job
    - **-npernode <number>**
    - used to indicate how many processor from each node needs to be allocated.
    - If omitted, the task manager will allocate all the process

Tips for FORTRAN Programmers

- Except **MPI_Wtime()** and **MPI_Wtick()** which are functions, all MPI Fortran subroutine differ from MPI C functions in that
  - The Fortran subroutine add one more argument at the end which is an integer that shows error status.
  - The C functions return an integer which shows error status
  - We don’t need to use special operator to differentiate parameter passing technique in Fortran.
  - MPI functions of C are case sensitive but MPI subroutines of Fortran are not
  - The C MPI function MPI_Initialize takes two argument which are not required in fortran MPI_Initialize subroutines

Tips for FORTRAN Programmers

- Example,if the MPI function statement
  - MPI_Comm_get_name(MPI_COMM_WORLD, comm_name, **&**comNameLength)**;**   is valid in C

  - **Call** MPI_Comm_get_name(MPI_COMM_WORLD, comm_name, comNameLength, **ierror**) - is valid in Fortran

Lab exercises

https://hpc-tutorials.llnl.gov/mpi/exercise_1/