C MPI Slurm Tutorial - Hello World

Introduction

The example shown here demonstrates the use of the Slurm Scheduler for the purpose of running a C/MPI program. Knowledge of C is assumed. Having read the Basic Slurm Tutorial prior to this one is also highly recommended.

Hello World!

MPI is a powerful foundation for your C programs which can enable you to write code that will run on not just one computer, but on multiple computers. By splitting your code up into more than one process, you will be able to do more work in a shorter amount of time. The Aries Cluster (which has MPICH installed) has this foundation in place and is the ideal place for running very complex or long-running programs.

The purpose of this tutorial is to show you the most basic C/MPI program possible - "Hello World" - and describe how it works and how to run it. Once you have mastered these basics, you will find more advanced material (such as the TSP program example) on the Star Lab website as well.

Part One: The Files

hello.tar

All the files needed to try out this example are available as a .tar file on the Star Lab website under the help heading. Under Linux, these files can be extracted with the following command:

```
tar xvf hello.tar
```

The files, summary

helloworld.c - The C program itself (includes main) which runs multiple processes, then prints out a "Hello World" message from each of those processes before exiting.
run.sh - A shell script which is designed to run with helloworld.c under the Slurm Scheduler.

Output Files, summary

slurm-XX.out - This file is generated by Slurm and shows the output and error information, if any. XX represents the slurm job number. May be used for troubleshooting.

mpi.out - Output of the helloworld program generated by run.sh

Part Two: Discussion of MPI in C

Before proceeding, a description of MPI's basic operation is in order. A C MPI program progresses through several stages before completion. In short, the stages can be summarized as follows:

Include mpi.h
Initialize the mpi environment (Parallel Code starts here)
Do the work and pass information between nodes by using MPI calls
Terminate the mpi environment (Parallel Code ends here)

MPI accomplishes this by sorting the available nodes into an object called a group. The group is referenced by another object which is tied to it - called a communicator. Although it is possible to use more than one communicator and set up multiple groups, for most purposes it is acceptable to do what we have done here - used just one communicator and one group. In particular, the communicator we have used is called MPI_COMM_WORLD (the predefined communicator) and it is associated with a group which contains all the MPI processes.

Any given process can be further identified by its rank. Every process within a particular group/under a particular communicator has a unique integer rank. These ranks are numbered starting at 0 and are contiguous. For example, if we are dealing with eight processes, they will be ranked from 0 to 7. In the event that a process is included in more than one group (this situation does not occur in the example program described here), it may have more than one rank associated with it. In a case such as this, the context is important - and it will be up to the programmer to specify exactly which group is being referred to at any given time (done by specifying the communicator).

In full, here is our example program, helloworld.c.

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

int main(int argc, char **argv)
{
  int rank, size;
  MPI_Init(&argc,&argv);

  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
```

```
  printf( "Hello world from process %d of %d\n",rank,size);
  MPI_Finalize();

  return 0;
}
```

Let us take a look at how the MPI mechanism operates in our example program as we move through main.

Let's assume that when helloworld.c was called, it was initialized to run using five processes. The program will ensure that five total processes have been generated when it executes the first MPI command - MPI_Init. Each of these processes has its own distinct set of program variables. For example, each process has its own rank and size variable.

Each process needs to know what its own rank (in this example, a number 0 - 4) and also needs to know the size of the group it is in (5 in this example since we have 5 processes). The MPI commands used for getting this information are MPI_Comm_rank and MPI_Comm_size.

Once a process has obtained this information, it will be able to complete the printf command that generates the "Hello World" message. As each process does this, it will also identify itself by rank, also showing the total number of processes when it reports.

The last MPI command called is MPI_Finalize, which terminates the execution environment.

More details regarding the different commands follow.

#include "mpi.h" - This include statement needs to go into every module that makes an MPI call.

int MPI_Init(int *argc, char ***argv) - Initializes the MPI execution environment. MPI_Init needs to be called prior to any other MPI functions, and therefore is required in every C program that uses MPI. MPI_Init must be called only once. (Note - MPI_Init obtains its arguments from the mpirun command. In our example, the mpirun command is included in the run.sh shell script - see Part Four. The mpirun command provides the number of nodes and a list of hosts to the MPI_Init command - see usage example below).

```
int main(int argc, char *argv[])
{
 …
 MPI_Init(&argc,&argv);
 …
}
```

int MPI_Comm_rank (MPI_Comm comm, int *rank) - Determines the rank of the calling process (and places it at the address *rank). (A rank can be anywhere from 0 up to the number of processors). The rank will also vary based upon the communicator used (in

our program, the only communicator used is MPI_COMM_WORLD). A process can have one rank under one communicator, but a different rank under another communicator.

int MPI_Comm_size (MPI_Comm comm, int *size) - Determines the number of processes in the current group of comm and places that integer at the address *size. In our example, the only communicator used is called MPI_COMM_WORLD.

MPI_Finalize(); - This command terminates the MPI execution environment. Therefore it needs to be the last MPI command called. This command needs to be included in every MPI program to ensure that all resources are properly released.

Part Three: Compilation

The executable can be created with a standard compilation command:

```
mpicc -o helloworld helloworld.c
```

Note that the mpicc command must be used when compiling a c program which depends upon mpi.

Execute the command above and generate the executable before continuing to Part Four.

NOTE - hello.tar also contains an example Makefile for this program which can be used in place of the command line compilation. Makefiles become especially important for programs that have more than one source file. The text of the example Makefile can also be viewed in the Appendix.

Part Four: Creating the Shell Script

There is a special method which must be used for running a C MPI program - simply typing the name of the executable is not enough! That is because, upon running the program, we need to specify additional details about the cluster.

The format of the mpiexec command is as follows:
```
mpiexec -hostfile <host_file> <executable> >> <output_file>
```

An example would be:
```
mpirun -hostfile $HOSTFILE ./helloworld >> ./mpi.out
```
Where $HOSTFILE is an automatically generated file from commands earlier in the script. Although it is theoretically possible to write the hostfile by hand as a text file, this should never be done. It is safer to have this done automatically by use of a shell script (as shown in run.sh below). This is because any error at all in the hostfile will cause the MPI program to crash (for example, in the event that the hostfile refers to a node that is offline).

The shell script works as a sort of super-executable. Not only does the shell script issue the mpirun command as described above, but it can also be made to generate the required host file on the fly.

An example shell script that works with the helloworld.c program is shown below. (Note that this shell script is designed to work only with Slurm — see Part Five for instructions on how to submit it to the scheduler).
This is run.sh:

```
#!/bin/bash

HOSTS=.hosts-job$SLURM_JOB_ID
HOSTFILE=.hostlist-job$SLURM_JOB_ID

srun hostname -f > $HOSTS
sort $HOSTS | uniq -c | awk '{print $2 ":" $1}' >> $HOSTFILE

echo "--------- The BEGINNING ---------" >> ./mpi.out
mpiexec -hostfile $HOSTFILE ./helloworld >> ./mpi.out

rm $HOSTS $HOSTFILE
```

Again, it is worth mentioning that this shell script will only run under Slurm (see Part Five for details on how to submit it to Slurm).

Also of note - we are attaching the output of the helloworld program to a file called "mpi.out." This file will be created if it does not already exist. This is necessary when using Slurm (sbatch) since the output would otherwise be lost.

Part Five: Submission of the Slurm Job

***Important Note**: The user must be logged into the ariessrv in order to submit a job successfully to Slurm.*

In order to ensure that jobs run in the fastest and most reliable manner possible, users must use the Slurm Scheduler to run jobs on the cluster. Slurm constantly monitors the nodes on the cluster and is able to track which nodes have the most resources free and which nodes are overburdened.

Slurm (sbatch) accepts a shell script (.sh file) which describes the job we want to run. We can use the shell script we created in Part Four. Note that this script must begin with the shebang line: `#!/bin/bash`

At this point, you must decide how many nodes and processes you would like to request for your job. Use the `sinfo` command to determine how many nodes are available or "idle" – since your job will not start to run until the requested amount of resources are

available (see final considerations in part 6 for more information). When choosing a number of processes, normally it is a good idea to choose no more than quadruple the number of nodes requested through slurm. For example, if ten machines are selected, the user would want to avoid choosing more than 40 processes. Performance will suffer, and in some cases slurm will refuse to run the job if this guideline is not followed. Run your MPI job with the following command (in this example, three nodes and 10 processes are requested):

```
sbatch --nodes=3 --ntasks=10 ./run.sh
```

Note that the number of processes is specified with the `ntasks` argument. Once the command has been issued, you will be given the job number. The `squeue` command can be issued at any time after submission in order to see where your job is in the queue. (See the Basic Slurm Tutorial for more details on how to track your job's progress, etc).

Once the job has finished, several new files will have been created. The slurm-XX.out file will provide troubleshooting information in the event that there was a problem with the execution.

Part Six: Final Considerations

For all long jobs (several hours or more) users are asked to use the Slurm Scheduler as described here out of fairness to other users and also as a way to ensure the most efficient possible use of Aries cluster resources.

It is also worth noting that when a user occupies more nodes, the program will, in theory, run faster. However, in practice, because Slurm will not start your job until the requested number of nodes become available, the same user may actually end up having to wait longer for the program to begin. Because of this trade-off, it is strongly suggested that users request only a moderate number of nodes when submitting a script. Not only will your program have the best chance to start running sooner, but some nodes will remain free for other users as well. Please be a good Star Lab citizen and do your part!

Appendix: Makefile for helloworld

A Makefile can make program compilation much easier — especially as the number of source files grows.

In order to compile the program, the user need only issue the command:

```
make
```

In order to clean up (remove the executable) the user need only issue this command:

```
make clean
```

This Makefile is made to work with helloworld and is provided for reference. (It can also be found in hello.tar)

```
CC =  mpicc

CFLAGS = -O3

All: helloworld

helloworld: helloworld.c
                $(CC) -o helloworld $(CFLAGS) helloworld.c

clean:
        rm helloworld
```