MPI Hello World

Author: Wes Kendall

In this lesson, I will show you a basic MPI hello world application and also discuss how to run an MPI program. The lesson will cover the basics of initializing MPI and running an MPI job across several processes. This lesson is intended to work with installations of MPICH2 (specifically 1.4). If you have not installed MPICH2, please refer back to the installing MPICH2 lesson.

> **Note** - All of the code for this site is on GitHub. This tutorial's code is under tutorials/mpi-hello-world/code.

Hello world code examples

Let's dive right into the code from this lesson located in mpi_hello_world.c. Below are some excerpts from the code.

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

int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
```

```
    MPI_Get_processor_name(processor_name, &name_len);

    // Print off a hello world message
    printf("Hello world from processor %s, rank %d"
           " out of %d processors\n",
           processor_name, world_rank, world_size);

    // Finalize the MPI environment.
    MPI_Finalize();
}
```

You will notice that the first step to building an MPI program is including the MPI header files with `#include <mpi.h>`. After this, the MPI environment must be initialized with:

```
MPI_Init(
    int* argc,
    char*** argv)
```

During `MPI_Init`, all of MPI's global and internal variables are constructed. For example, a communicator is formed around all of the processes that were spawned, and unique ranks are assigned to each process. Currently, `MPI_Init` takes two arguments that are not necessary, and the extra parameters are simply left as extra space in case future implementations might need them.

After `MPI_Init`, there are two main functions that are called. These two functions are used in almost every single MPI program that you will write.

```
MPI_Comm_size(
    MPI_Comm communicator,
    int* size)
```

`MPI_Comm_size` returns the size of a communicator. In our example, `MPI_COMM_WORLD` (which is constructed for us by MPI) encloses all of the

processes in the job, so this call should return the amount of processes that were requested for the job.

```
MPI_Comm_rank(
    MPI_Comm communicator,
    int* rank)
```

`MPI_Comm_rank` returns the rank of a process in a communicator. Each process inside of a communicator is assigned an incremental rank starting from zero. The ranks of the processes are primarily used for identification purposes when sending and receiving messages.

A miscellaneous and less-used function in this program is:

```
MPI_Get_processor_name(
    char* name,
    int* name_length)
```

`MPI_Get_processor_name` obtains the actual name of the processor on which the process is executing. The final call in this program is:

`MPI_Finalize` is used to clean up the MPI environment. No more MPI calls can be made after this one.

Running the MPI hello world application

Now check out the code and examine the code folder. In it is a makefile.

```
>>> git clone https://github.com/wesleykendall/mpitutorial
>>> cd mpitutorial/tutorials/mpi-hello-world/code
>>> cat makefile
EXECS=mpi_hello_world
MPICC?=mpicc

all: ${EXECS}
```

```
mpi_hello_world: mpi_hello_world.c
    ${MPICC} -o mpi_hello_world mpi_hello_world.c

clean:
    rm ${EXECS}
```

My makefile looks for the MPICC environment variable. If you installed MPICH2 to a local directory, set your MPICC environment variable to point to your mpicc binary. The mpicc program in your installation is really just a wrapper around gcc, and it makes compiling and linking all of the necessary MPI routines much easier.

```
>>> export MPICC=/home/kendall/bin/mpicc
>>> make
/home/kendall/bin/mpicc -o mpi_hello_world mpi_hello_world.c
```

After your program is compiled, it is ready to be executed. Now comes the part where you might have to do some additional configuration. If you are running MPI programs on a cluster of nodes, you will have to set up a host file. If you are simply running MPI on a laptop or a single machine, disregard the next piece of information.

The host file contains names of all of the computers on which your MPI job will execute. For ease of execution, you should be sure that all of these computers have SSH access, and you should also setup an authorized keys file to avoid a password prompt for SSH. My host file looks like this.

```
>>> cat host_file
cetus1
cetus2
cetus3
cetus4
```

For the run script that I have provided in the download, you should set an environment variable called MPI_HOSTS and have it point to your hosts

file. My script will automatically include it in the command line when the MPI job is launched. If you do not need a hosts file, simply do not set the environment variable. Also, if you have a local installation of MPI, you should set the MPIRUN environment variable to point to the mpirun binary from the installation.

Once this is done, you can use the run.py python script that is included in the main repository. It is stored under the *tutorials* directory and can execute any program in all of the tutorials (it also tries to build the executables before they are executed). Try the following from the root mpitutorial folder.

```
>>> export MPIRUN=/home/kendall/bin/mpirun
>>> export MPI_HOSTS=host_file
>>> cd tutorials
>>> ./run.py mpi_hello_world
/home/kendall/bin/mpirun -n 4 -f host_file ./mpi_hello_world
Hello world from processor cetus2, rank 1 out of 4 processors
Hello world from processor cetus1, rank 0 out of 4 processors
Hello world from processor cetus4, rank 3 out of 4 processors
Hello world from processor cetus3, rank 2 out of 4 processors
```

As expected, the MPI program was launched across all of the hosts in my host file. Each process was assigned a unique rank, which was printed off along with the process name. As one can see from my example output, the output of the processes is in an arbitrary order since there is no synchronization involved before printing.

Notice how the script called mpirun. This is program that the MPI implementation uses to launch the job. Processes are spawned across all the hosts in the host file and the MPI program executes across each process. My script automatically supplies the *-n* flag to set the number of MPI processes to four. Try changing the run script and launching more processes! Don't accidentally crash your system though. :-)

Now you might be asking, *"My hosts are actually dual-core machines. How can I get MPI to spawn processes across the individual cores first before individual machines?"* The solution is pretty simple. Just modify your hosts file and place a colon and the number of cores per processor after the host name. For example, I specified that each of my hosts has two cores.

```
>>> cat host_file
cetus1:2
cetus2:2
cetus3:2
cetus4:2
```

When I execute the run script again, *voila!*, the MPI job spawns two processes on only two of my hosts.

```
>>> ./run.py mpi_hello_world
/home/kendall/bin/mpirun -n 4 -f host_file ./mpi_hello_world
Hello world from processor cetus1, rank 0 out of 4 processors
Hello world from processor cetus2, rank 2 out of 4 processors
Hello world from processor cetus2, rank 3 out of 4 processors
Hello world from processor cetus1, rank 1 out of 4 processors
```

Up next

Now that you have a basic understanding of how an MPI program is executed, it is now time to learn fundamental point-to-point communication routines. In the next lesson, I cover basic sending and receiving routines in MPI. Feel free to also examine the MPI tutorials for a complete reference of all of the MPI lessons.

Having trouble? Confused? Feel free to leave a comment below and perhaps I or another reader can be of help.

This site is hosted entirely on GitHub. This site is no longer being actively

contributed to by the original author (Wes Kendall), but it was placed on GitHub in the hopes that others would write high-quality MPI tutorials. Click here for more information about how you can contribute.