Compiling and Running Open MPI Programs

Stewart Weiss.

Compiling an MPI Program

Use `mpicc` to compile C programs, `mpic++` to compile C++ code.
To compile a program using `mpicc`, use the same options as you would for `gcc` but use `mpicc` instead. For example, to build an executable from a single source file named `my_mpi_prog.c`, turning on all warnings (`-Wall`) and the debugging symbols (`-g`) use

```
mpicc -Wall -g -o my_mpi_prog  my_mpi_prog.c
```

Running an MPI Program on a Single Host

To run an executable that has been compiled using `mpicc` or `mpic++`, use `mpirun` with the appropriate flags. See the **OpenMPI** man page for a complete list of the flags and options. This **README** document only describes some basic running options.

- To schedule MPI processes on separate cores on a single multi-core processor use

  ```
  mpirun --bind-to core -np <NUMCORES>  <executable>
  ```

  where < `NUMCORES` > is the number of cores on the processor and < `executable` > is the name of your executable MPI program.
  To get the number of cores on the processor, one easy command is:

  ```
  lscpu | grep "^CPU(s):"
  ```

  whose output on a 12 core host will be

```
CPU(s):              12
```

To get just the number, use

```
lscpu | grep -w "^CPU(s)" | awk -F":[ ]*"  '{print $2}'
```

which will output

```
12
```

- The `-np` option is how you specify the number of processes to create.
  If you just type

  ```
  mpirun -np N  <executable>
  ```

  MPI will schedule `N` tasks across however many cores it finds on your local host, and try to load balance accordingly. For example, 12 processes on 4 cores will result in 3 per core on average.

- To attach information about which process wrote output, use

  ```
  mpirun -tag-output -np <NUMCORES>  <executable>
  ```

  which prepends `[jobid, rank]<stdxxx>` to each output line, where `stdxxx` is either `stderr` or `stdout`.

Scheduling MPI Processes on Multiple Hosts

This is a complex topic, but the general idea is that MPI has to be able to launch your processes on the separate hosts, and to do that it needs

1. to know the addresses of the hosts on which to run them, and

2. to be able to start up processes using your username on the remote hosts.

The first requirement is solved by giving it a *hostfile*, as is shown below.
The second is solved on by using **ssh authentication** keys as a means of authentication instead of passwords. Instructions for how to set up ssh authentication using an RSA key-pair are below, site specific to our Computer Science Department network.

Setting Up SSH Authentication

One Time Configuration. Set up SSH as the authentication agent as follows:

To create authentication keys, type the following three commands on any `cslab` host:

```bash
 ssh-keygen -t rsa
 cp  ~/.ssh/id_rsa.pub   ~/.ssh/authorized_keys
 chmod go-rwx ~/.ssh/authorized_keys
```

The first command creates the public and private RSA keys, storing the public key in `~/.ssh/id_rsa.pub`; the second copies the public key to the file `~/.ssh/authorized_keys`; the third removes read/write/execute permission from everyone except the owner of the file,i.e., you.

Because your home directory is remotely mounted by every host in the lab, all hosts on which you would launch an MPI process will have a copy of the public key in the `authorized_keys` file, and all from which you might launch a job will have the private key in `~/.ssh/id_rsa`, making the key-pair authentication universal among the set of lab hosts.

Per Session Configuration. Each time you start a new work session, meaning each time you login, type the following:

```bash
 ssh-agent bash
 ssh-add
```

You can create a script if you like to make this one step. Put the following lines into a file named `startsession`, and make it executable:

```bash
 #!/bin/bash
```

```
ssh-agent bash
ssh-add
```

Then just type `startsession` once after logging in.
These lines basically make `bash` an agent for ssh, storing your private key(s) in the agent for authentication.

Creating a Hostfile

A *hostfile* is a file that contains, int its simplect form, one host IP address per line. It can also contain how many **slots**, i.e., cores, that host has. You can also use DNS names instead of IP addresses.

The script on the server in the `mpi_demos/scripts` directory named `buildHostfile.sh` is designed to create a list on standard output of all available hosts named in the form `cslabXX` in the Computer Science Department network. To create a hostfile, run it and redirect output to a file. Enter the command

```
buildHostfile > myhostfile
```

It will also print a message on the terminal stating how many slots and hosts are available. Save the number of slots that it outputs.

Running the MPI Program Using the Hostfile

You can now run your MPI program using this hostfile. Make sure that the number of processes specified by the `-np` option does not exceed the total number of slots. It can be smaller. Run the command

```
mpirun --hostfile myhostfile -np N <executable>
```

where `N` is the number of processes to run on the hosts and < `executable` > is the name of the program. If there are errors, it may be because some of the hosts are no longer accessible when you run the program. In this case you can edit the hostfile to remove the hosts that MPI reported as unavailable.

Debugging an MPI Program

Debugging any parallel program is difficult. There are two types of errors, or "bugs":

- Timing-independent errors
- Timing-dependent errors

The first are errors that result in incorrect output or failures regardless of the relative order in which the processes execute. They are easier to find than the second ones.

Timing-dependent errors depend on the relative order in which the processes execute. They can be extremely hard to find.

Regardless of which type your bug is, the debugging procedure should begin in the same way. The procedure that I recommend is what follows.

1. Modify the source code of your main program by including the following code and comments in a place where you want to start stepping through the code, such as immediately after the program initializes the MPI library in `main()`:

```
#ifdef DEBUG_ON
    /* To debug, compile this program with the -DDEBUG_ON option,
       which defines the symbol DEBUG_ON,  and run the program as usual
       with mpirun.
       When the output appears on the terminal, listing the pids of the
       processes and which hosts they are on, choose the lowest
       pid P on the machien you are connected to.
       Open a new terminal window and in that window issue the command
           gdb --pid P
           (or gdb -p P on some systems)
       and after gdb starts, go up the stack to main by entering the
       command
           up 3
      (main will be three stacks frames above your current frame,
       which should be nanosleep.)
      Then enter the command
           set var i = 1
      to break the while loop. You can now run ordinary gdb commands to
      debug this process. This should be process 0.
      Repeat these steps for each other process that you created in
      the mpirun command.
    */
    i = 0;
    char hostname[256];
    gethostname(hostname, sizeof(hostname));
    printf("PID %d on %s ready for attach\n", getpid(), hostname);
    fflush(stdout);
```

```
        while (0 == i)
            sleep(5);
    #endif
```

2. The comment above describes what you must do, but to clarify some of the steps: After you insert the code, compile with a line of the form

   ```
   mpicc -g -DDEBUG_ON -Wall -o <your_executable>   <source_files>
   ```

   If you need to pass linker or loader flags you do that as well.

3. Run the program with the least number of processes you need to look for the bug. I suggest two to start.

4. After you start stepping through each process, if you reach a communication point or a barrier, you'll have to make sure all processes reach it before the others can continue.

5. If you are unfamiliar with `gdb`, now is the time to learn it.