CS3210 – 2016/17 Semester I

Lab 4

Introduction to Distributed-Memory Programming using MPI

Objectives:

1. Learn distributed-memory parallel programming via MPI with C/C++
2. Learn how to compile and run an MPI program
3. Learn how to map MPI processes on different number of nodes and cores

MPI (Message Passing Interface) is a standardized and portable programming toolkit for programming distributed-memory systems. It is supported by a few programming languages as an external library. In our lab, we use MPI with C/C++.

Note: All files in this lab can be found on IVLE workbin, or directly downloaded via wget:

```
$ wget www.comp.nus.edu.sg/~sooyj/cs3210/Lab4/<filename>

e.g.
$ wget www.comp.nus.edu.sg/~sooyj/cs3210/Lab4/mm-mpi.c
```

1. Compiling and Running MPI Programs

An MPI program consists of multiple processes cooperating via series of MPI routine calls. Each MPI process has a unique identifier called ***rank***. Processes are grouped into sets called ***communicators***. By default, all MPI processes in a program belong to a communicator called **`MPI_COMM_WORLD`**.

Let's look at the **`hello.c`** to identify the basic elements of an MPI program:

```
$ vim hello.c
```

To compile this program, use the command `mpicc`:

```
$ mpicc hello.c -o hello
```

To run the program, use the command `mpirun`:

```
$ mpirun -np 4 ./hello
```

`mpirun` is a script that receives at least two parameters:

1. The number of processes to be created, **`-np <n>`**, where **`n`** is an integer.
2. The path to the program binary. In this case, **`./hello`**

Exercise 1: Run the program **locally**, the node where the program is downloaded, with 1, 4 and 32 processes and observe the output. Comment on the ordering of "**`hello world`**" messages.

2. Processes to Processors Mappings

Next, we compile another MPI program. We will run this program using two nodes called **`soctf-pdc-001`** and **`soctf-pdc-002`** (**you need to replace the node names with the hostnames of the computers in your local cluster**).

Exercise 2: Compile the program **`loc.c`**. If you compile the program **`on soctf-pdc-001`** then run the program on the remote node **`soctf-pdc-002`**, or vice-versa.

```
$ mpirun -H soctf-pdc-002 -np 16 ./loc
```

The **`-H soctf-pdc-002`** argument allows us to specify the node to run our MPI processes.

Notice that `mpirun` asks for your password, and displays an error:

```
--------------------------------------------------------------------------
mpirun was unable to launch the specified application as it could not
access or execute an executable:

Executable: ./loc
Node: soctf-pdc-002

while attempting to start process rank 0.
--------------------------------------------------------------------------
```

This error occurs because the binary code is not found in the remote node, and you need to first copy the program binary to the remote host before execution.

Exercise 3: Using the instructions from Lab 2, setup your SSH keys to connect to the remote host without typing your password.

Exercise 4: Copy the **`loc`** binary to the remote host and run the program.

The program outputs the location of each process that is started together with the processor affinity mask. MPI allows us to specify exactly where we want the processes to execute. The easiest way to accomplish this is to use a machine configuration file. This file contains a list of hostnames of the nodes where your processes will run. For example, you can create a file **`machinefile.1`** as follows:

```
$ vim machinefile.1
soctf-pdc-001
soctf-pdc-002
```

specifies that process 0 starts on node `soctf-pdc-001` and process 1 on `soctf-pdc-002`. If there are more than two processes, `mpirun` will cycle through the listed nodes in a round-robin fashion by default.

```
$ mpirun -machinefile machinefile.1 -np 4 ./loc
Process 0 is on hostname soctf-pdc-001 on processor mask 0xff
Process 1 is on hostname soctf-pdc-002 on processor mask 0xff
Process 2 is on hostname soctf-pdc-001 on processor mask 0xff
Process 3 is on hostname soctf-pdc-002 on processor mask 0xff
```

A more precise control of the mapping can be achieved using a machine file and a rank file. The rank file lists each MPI rank number, starting with 0 and tells exactly which nodes, which sockets and which cores can be used by that rank. The rank file lists each rank on a line, with the following syntax:

rank <*number*>=<*hostname*> slot=<*socket_range*>:<*core_range*>

For example, download the sample rank file **rankfile.1** and take a look.

```
$ vim rankfile.1
rank 0=soctf-pdc-001 slot=0:0
rank 1=soctf-pdc-002 slot=0:0
rank 2=soctf-pdc-001 slot=0:0
rank 3=soctf-pdc-002 slot=0:0-2
rank 4=soctf-pdc-002 slot=0:0-2
rank 5=soctf-pdc-002 slot=0:0-2
```

specifies that process 0 is mapped to node **soctf-pdc-001** on **core 0** of **socket 0**, then process 1 is mapped to node **soctf-pdc-002 socket 0 core 0**, process 2 on node **soctf-pdc-001 socket 0 core 0** and then processes 3, 4 and 5 will be started on node **soctf-pdc-001** on **socket 0**, and are free to be executed on either of cores **0, 1 or 2.**

Here is an example of using **rankfile.l**

```
$ mpirun -machinefile machinefile.1 -rankfile rankfile.1 -np 6 ./loc
Process 2 is on hostname soctf-pdc-001 on processor mask 0x1
Process 1 is on hostname soctf-pdc-002 on processor mask 0x1
Process 5 is on hostname soctf-pdc-002 on processor mask 0x7
Process 3 is on hostname soctf-pdc-002 on processor mask 0x7
Process 0 is on hostname soctf-pdc-001 on processor mask 0x1
Process 4 is on hostname soctf-pdc-002 on processor mask 0x7
```

For more details on mapping between processes and cores, consult `mpirun`'s manual.

Exercise 5: We have a total of 12 cores in your workbench. Run the program **loc** with 12 processes such that it maps each process to one core. Show your TA once you get it working.

Exercise 6: Download and compile the MPI matrix multiplication program, **mm-mpi.c**. Create and test the following four MPI configurations:

1. Using 4 processes, all on the Core i5 node, binding each process to one core.
2. Using 4 processes, all on the Core i5 node, without process binding.
3. Using 8 processes, all on the Core i7 node, without process binding.
4. Using 12 processes, on both nodes, without process binding.