Lab 4
Introduction to Distributed-Memory Programming using MPI

CS3210 – 2021/22 Semester 1

Learning Outcomes

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
4. Learn blocking and non-blocking process-to-process communication

MPI (Message Passing Interface) is a standardized and portable programming toolkit for programming distributed-memory systems using message passing. It is supported by a few programming languages as an external library (there are many different implementations of the MPI standard).

MPI Standards and Implementations

MPI (Message Passing Interface) is a **message passing library standard** based on the consensus of the MPI Forum which comprises many commercial and research organizations and personnel. The MPI Forum first released the MPI-1 standard in 1992, which was followed by MPI-2 (2000), MPI-2.1 (2008), MPI-2.2 (2009), MPI-3 (2012), MPI-3.1 (2015), and MPI-4 (2021).

MPI-1: established the basic standards required for message passing between nodes. Later versions of the standard introduced more features.

MPI-2: introduced Dynamic Processes, One-Sided Communication including shared memory operations, Extended Collective Operations, External Interfaces to allows developers to add features such as debuggers, C++ Bindings, and parallel I/O.

MPI-3: introduced Non-blocking Collective Operations (such as `MPI_Iscatter`), Neighborhood Collectives for use with virtual topologies, Fortran Bindings, and Performance Tools.

More information about the MPI standard can be found here: MPI Forum documents.

Please do not confuse the MPI standard with MPI Implementations. The MPI standard defined by the MPI Forum as explained above is implemented differently by different organizations. Without these implementations, we cannot use MPI routines in our programs. Some of such MPI implementations include MPICH, IBM-MPI, and, OpenMPI, among others. In our labs we are using OpenMPI, which is an open-source implementation of the MPI standard.

OpenMPI version 4.0.3 (with full support for the MPI-3 standard) has been installed on the lab machines. You can print the version information by running `ompi_info` in the command line (terminal) - note that the version number of OpenMPI is separate from the version of the MPI standard that is implemented.

Part 1: Compiling and Running MPI Programs

An MPI program consists of multiple processes cooperating via a series of MPI routine calls. Each MPI process has a unique identifier known as the *rank*. Ranks are used by the programmer to specify the source and destination of messages as well as to conditionally control the program execution (ie. `if (rank == 0) { // do this } else { // do that }`). Processes are grouped into sets called communicators. By default, all MPI processes in a program belong to the global communicator MPI_COMM_WORLD.

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

- Open the code in a terminal (console). You may use any available text editor:
  `> vim hello.c`
- To compile this program, use the command `mpicc`:
  `> mpicc hello.c -o hello`
- To run the program with 4 processes, use the command `mpirun`:
  `> mpirun -np 4 ./hello`

`mpirun` is a script that accepts 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.

Part 2: Mapping MPI Processes to Processors

Next, we compile another MPI program. We will run this program using two nodes called `soctf-pdc-001` and `soctf-pdc-009`. **Replace these hostnames here with those assigned to you.**

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-009`, or vice-versa.

- We can specify the node on which we need to run our MPI processes. In the below command, using the `-H` flag directs MPI to run MPI processes on `soctf-pdc-009`.
  `> mpirun -H soctf-pdc-009 -np 1 ./loc`

Notice that the program hangs after executing `mpirun`. This is because ssh-passwordless access is not set up between the local and the remote node.

Exercise 3

Using the instructions from Tutorial 1, setup your SSH keys to connect to the remote host without typing your password.

Now, try to rerun the MPI program using `mpirun`. You should see the following error:

```
--------------------------------------------------------------------------
mpirun was unable to find the specified executable file, and
therefore did not launch the job.  This error was first reported for
process rank 0; it may have occurred for other processes as well.

NOTE: A common cause for this error is misspelling a mpirun command
      line parameter option (remember that mpirun interprets the
      first unrecognized command line token as the executable).

Node:       soctf-pdc-009
Executable: loc
--------------------------------------------------------------------------
```

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

Note that in this command we use the `-np 1` flag. When specifying the host with the `-H` flag, OpenMPI by default assumes that there is only one processing slot per node. Thus, if we try to run with a larger number of processes, we will get an error. (This does not happen when a machinefile is used as you will learn in the next section. Alternatively, you can use `--host` and specify the number of slots of a host via a `":N"` suffix on the hostname, where N is the number of slots, e.g. `mpirun --host soctf-pdc-009:4 -np 3 ./loc`.)

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 or machinefile (also known as hostfile in the documentation). The machinefile contains a list of hostnames of the nodes where your processes will run, and optionally the number of slots on each node. **A slot is simply a reserved place for an MPI process, and does not correspond to a hardware resource like a core.** For example, you can create the file `machinefile.1` as follows:

```
soctf-pdc-001 slots=6
soctf-pdc-009
```

The above specifies that `soctf-pdc-001` and `soctf-pdc-009` are used to execute MPI processes, and that there are 6 slots on `soctf-pdc-001`. By default, each slot can only be assigned 1 MPI process. **If the**

machinefile does not specify the number of slots on a node, OpenMPI assumes it is the number of physical cores on that node.

By default (if the `--map-by <option>` flag is not specified), OpenMPI assigns processes to slots by exhausting all slots on a host before proceeding to the next host in the machinefile. Thus, for the above example, processes 0 to 5 will start on `soctf-pdc-001` and processes 6 to 9 will start on `soctf-pdc-009` (which has only 4 physical cores).

Note that if `-np` parameter is higher than the total number of slots on all nodes, you will get an error. However, you can force OpenMPI to allow each slot to accept more than 1 process with the `--oversubscribe` flag. Read more about this and process placement here.

You can change the process-slot mapping policy by specifying the `--map-by <policy>` flag. For example, specifying `--map-by node` will let OpenMPI assign processes to slots from different hosts in a round-robin (load balancing) fashion. For the above example, this means processes 0, 2, 4, 6, 8 and 9 will start on `soctf-pdc-001` whilst processes 1, 3, 5, 7 will start on `soctf-pdc-009`.

- Execute `loc` with `mpirun` while specifying the relevant machinefile with the `slot` mapping policy. Remark that this is similar to the default (`socket`) mapping policy, except when one or more hosts contain multiple CPU sockets.
  `> mpirun -machinefile machinefile.1 -np 10 --map-by slot ./loc | sort`

This command will generate the following output (`sort` is used to sort the output text in alphabetical order):

```
Process 0 is on hostname soctf-pdc-001 on processor mask 0xffff
Process 1 is on hostname soctf-pdc-001 on processor mask 0xffff
Process 2 is on hostname soctf-pdc-001 on processor mask 0xffff
Process 3 is on hostname soctf-pdc-001 on processor mask 0xffff
Process 4 is on hostname soctf-pdc-001 on processor mask 0xffff
Process 5 is on hostname soctf-pdc-001 on processor mask 0xffff
Process 6 is on hostname soctf-pdc-009 on processor mask 0xff
Process 7 is on hostname soctf-pdc-009 on processor mask 0xff
Process 8 is on hostname soctf-pdc-009 on processor mask 0xff
Process 9 is on hostname soctf-pdc-009 on processor mask 0xff
```

Compare the above to the example below to understand the difference between the `slot` and `node` policies for mapping MPI processes to available slots.

- Execute `loc` with `mpirun` while specifying the relevant machinefile with the `node` mapping policy.
  `> mpirun -machinefile machinefile.1 -np 10 --map-by node ./loc | sort`

This command will generate the following output:

```
Process 0 is on hostname soctf-pdc-002 on processor mask 0xff
Process 1 is on hostname soctf-pdc-001 on processor mask 0xffff
Process 2 is on hostname soctf-pdc-002 on processor mask 0xff
Process 3 is on hostname soctf-pdc-001 on processor mask 0xffff
Process 4 is on hostname soctf-pdc-002 on processor mask 0xff
Process 5 is on hostname soctf-pdc-001 on processor mask 0xffff
Process 6 is on hostname soctf-pdc-002 on processor mask 0xff
Process 7 is on hostname soctf-pdc-001 on processor mask 0xffff
Process 8 is on hostname soctf-pdc-001 on processor mask 0xffff
Process 9 is on hostname soctf-pdc-001 on processor mask 0xffff
```

Precise control of the mapping can be achieved by providing `mpirun` with a rankfile instead of a machinefile. The rankfile specifies for each MPI process of rank i (starting with 0), which host it executes on, and which socket and logical cores it is bound to. (Note that for MPI jobs, a process' `rank` refers to its rank in the communicator `MPI_COMM_WORLD`). The rank file lists each rank on a line, with the following syntax:

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

> Note that *mapping* only assigns a default location to each MPI process on a host - the process can migrate between different cores on that host during execution. In contrast *binding* explicitly restricts the set of *logical* cores an MPI process can execute on, using the processor affinity bitmask.

For example, download the sample rank file `sample-rankfile` and take a look (shown below).

```
rank 0=soctf-pdc-001 slot=0:0
rank 1=soctf-pdc-009 slot=0:0
rank 2=soctf-pdc-001 slot=0:0
rank 3=soctf-pdc-009 slot=0:0-2
rank 4=soctf-pdc-009 slot=0:0-2
rank 5=soctf-pdc-009 slot=0:0-2
```

The above 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-009` 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-009` on socket 0, and are free to be executed on either of cores 0, 1 or 2. **Note that the rankfile uses physical core indices - if the system implements hardware multi-threading, binding a process to core 0 will bind it to more than one logical core.**

Shown below is an example of how `sample-rankfile` is used.

```
> mpirun -rankfile sample-rankfile -np 6 ./loc | sort
```

This command generates the following output:

```
Process 0 is on hostname soctf-pdc-001 on processor mask 0x401
Process 1 is on hostname soctf-pdc-009 on processor mask 0x11
Process 2 is on hostname soctf-pdc-001 on processor mask 0x401
Process 3 is on hostname soctf-pdc-009 on processor mask 0x77
Process 4 is on hostname soctf-pdc-009 on processor mask 0x77
Process 5 is on hostname soctf-pdc-009 on processor mask 0x77
```

For more details on mapping, ranking and binding, consult the OpenMPI documentation for `mpirun`.

More information and further reading:

- OpenMPI v4.0 official documentation: https://www.open-mpi.org/doc/v4.0/
- LLNL MPI Tutorial: https://computing.llnl.gov/tutorials/mpi/
- OpenMPI FAQ: https://www.open-mpi.org/faq/?category=running
- Another MPI Tutorial: http://mpitutorial.com/

Exercise 5

We have two machines allocated to you with a total of at least 14 cores. Run the program `loc` with 14 processes such that it maps each process to one (unique) core.

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 i7 node, binding each process to one core.
2. Using 4 processes, all on the Core i7 node, without process binding.
3. Using 10 processes, all on the Core Xeon node, without process binding.
4. Using 14 processes, on both nodes, without process binding.

Describe the type of data distribution currently used in the matrix multiplication problem. Explain advantages and disadvantages to this distribution. You may use performance measurements to prove your points. Change the implementation to use another data distribution type. Try to choose a distribution that might translate into a better or (roughly) similar performance (speedup).

Part 3: Process-to-process Communication

MPI provides two types of process-to-process communication: blocking and non-blocking. Each of these communication types is accomplished with a send and a receive function.

Part 3.1: Blocking Communication

Blocking send stops the caller until the message has been copied over to the underlying communication buffer (i.e. network buffer). Similarly, blocking receive blocks the calling process until the message has been received in the MPI process.

The function signatures for a blocking send and receive are respectively:

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
           int source, int tag, MPI_Comm comm, MPI_Status *status)
```

Table 1 lists the arguments to be used with blocking communication functions.

Table 1: Arguments for Blocking Communication Routines

buf	Pointer to the memory buffer that holds the contents of the message to be sent or received
count	The number of items that will be sent.
datatype	Specifies the primitive data type of the individual item sent in the message, and can be one of the following: MPI_CHAR MPI_SHORT MPI_INT MPI_LONG MPI_UNSIGNED_CHAR MPI_UNSIGNED_SHORT MPI_UNSIGNED_LONG MPI_UNSIGNED MPI_FLOAT MPI_DOUBLE MPI_LONG_DOUBLE MPI_BYTE MPI_PACKED
dest/source	Specifies the rank of the source / destination process in that MPI communicator
tag	An integer that allows the receiving process to distinguish a message from a sequence of messages originating from the same sender
comm	The MPI communicator
status	Pointer to an MPI_STATUS structure that allows us to check if the receive has been successful.

Exercise 7

Compile the program `block_comm.c` and run it with two processes.

Exercise 8

Modify the file `block_comm.c` (new name `block_comm_1.c`) such that process 1 sends back to process 0 ten floating-point values in one message. Compile the program and run it.

Exercise 9

What happens if we flip the order of `MPI_Send` and `MPI_Recv` in the master process? Discuss the implication.

Part 3.2: Non-blocking Communication

Non-blocking communication, in contrast to blocking communication, does not block either the sender or the receiver.

The function signatures for a non-blocking send and receive are respectively:

```
int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm, MPI_Request *request)
```

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
          int source, int tag, MPI_Comm comm, MPI_Request *request)
```

The only new parameter in the call is:

request	Pointer of type MPI_Request which provides a handle to this operation. Using this handle, the programmer can inquire later whether the communication has completed.

Both functions return immediately, and the communication will be performed asynchronously w.r.t. the rest of the computation. Using these functions, the MPI program can overlap communication with computation. It is unsafe to modify the application buffer (your variable) until you know for a fact the requested non-blocking operation was actually performed by the library.

To check whether the functions have finished communicating, we can use:

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

MPI_Test takes a handle of an `MPI_Isend` or `MPI_Irecv` and stores a true/false in the flag variable which indicates whether the operation has been completed.

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

`MPI_Wait` blocks the current process until the request has finished.

MPI provides quite a few variants of these basic two functions. You are encouraged to read the MPI manual for these functions:

```
MPI_Test, MPI_Testall, MPI_Testany, MPI_Testsome
MPI_Wait, MPI_Waitany, MPI_Waitsome
```

MPI does NOT guarantee fairness, thus the programmer should make sure that starvation does not occur.

Exercise 10

Compile the program `nblock_comm.c` and run it with 3 processes. Modify the source (new name `nblock_comm_1.c`) code to swap the order of the `MPI_Isend` and `MPI_Irecv`. Compile and run it again. What do you observe?

More information and further reading:

- Communication in a ring: Correctness and Performance. https://www.epcc.ed.ac.uk/blog/2019/what-mpi-nonblocking-correctness-and-performance

Lab sheet (2% of your final grade):
You are required to produce a write-up with the results for exercises 6, 9, and 10. Submit the lab report (in PDF form with file name format A0123456X.pdf) via LumiNUS before Fri, 22 Oct. The document must contain

- explanation on how you have solved each exercise (maximum 3 paragraphs in total)
- results and the raw measured data for exercise 6 - use graph(s) and try to justify your observations

Write down the hostnames of the nodes you used for your experiments. When you run your experiments, check if any other student is using the same node. You might experience varying results when someone else runs their programs on the same node.

The upcoming assignment 3 will be on MPI. To alleviate the potential resource contention between students, you can use Slurm to submit MPI executable to execute during the blackout period. Scheduling your MPI executable to run during the blackout period helps to ensure the accuracy of the measured performance of your MPI executable since the executable will run exclusively on requested nodes and would not be affected by the programs of other students. You can refer to Slurm Student User Guide on how to submit MPI jobs to run in the lab machines using Slurm.