Lab 4
Introduction to Distributed-Memory Programming using MPI

CS3210 – 2024/25 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 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) (see the MPI Forum documents for more details).

Please do not confuse the MPI standard with MPI Implementations. The MPI standards above are implemented differently by different organizations such as 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 4.1.2** (with full support for the MPI-3 standard) is installed on the lab machines. You can print the version information by running `ompi_info` in the command line (terminal).

Logging in & Getting Started

Follow the instructions from previous (non-GPU) labs / tutorials to `ssh` into one of our lab machines. Remember that you can still refer to https://nus-cs3210.github.io/student-guide/. Now, download the code from Canvas or https://www.comp.nus.edu.sg/~srirams/cs3210/L4_code.zip

Important: Telling MPI Which Network Interface to Use

MPI needs to know which network interface to send messages on, and our `soctf` machines have multiple. Please use **one** of these options (**otherwise this lab will not work!**)

1. Add the option `--mca btl_tcp_if_include 172.26.187.34/23` to each `mpirun`
2. Run `export OMPI_MCA_btl_tcp_if_include=172.26.187.34/23` in each terminal you use, or add the line to your `.bashrc` or `.bash_profile`

Part 1: Compiling and Running MPI Programs Locally

An MPI program consists of multiple processes cooperating via a series of MPI routine calls. Each MPI process has a unique identifier known as its *rank*. Ranks are used by the programmer to specify the source and destination of messages as well as to conditionally control the program execution:
(e.g., `if (rank == 0) { // do this } else { // do that }`). Processes can communicate with each other using *communicators*. By default, all MPI processes in a program have access to the global communicator MPI_COMM_WORLD.

Let's look at the `hello.cpp` 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.cpp
  ```

- **Study the MPI calls in `hello.cpp` (those that start with `MPI_`).**
- To compile this program, use the command `mpic++`:

  ```
  > mpic++ hello.cpp -o hello
  ```

- To run the program with 4 processes, use the command `mpirun`:

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

`mpirun` is a script that executes MPI programs. We use two of its 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

Compile and run the `hello.cpp` program locally (i.e., without Slurm) on an soctf lab machine as shown above, with 1, 4 and 32 processes, and observe the output.

- How are the "hello world" messages ordered?
- Is there a limit to the number of MPI processes you can launch on your single node? What if you use the `--oversubscribe` option?

Let's try to quickly run our MPI code on multiple nodes "manually".

Exercise 2

- Make sure your SSH key is copied to `soctf-pdc-001` and 002 for this exercise. If not, run: `ssh-copy-id soctf-pdc-001` (it'll be present in the other nodes automatically due to NFS)
- Now run the following command to run `hello` on `soctf-pdc-001` and 002: `mpirun -np 4 -H soctf-pdc-001:2,soctf-pdc-002:2 ./hello`

  Notice that we distributed the tasks across the two hosts we specified.

Part 2: Running MPI Programs across Multiple Nodes

However, the previous exercises do not show what makes MPI interesting, which is its ability to run programs **across many nodes, easily!**

Using MPI + Slurm

Let's run our MPI program with Slurm!. Slurm will allocate the resources we request, and `mpirun` will automatically detect the resources allocated to our job, and run it for us. However, `mpirun` cannot work directly with `srun`. We will need to use a new command, `salloc`, if we want to run MPI jobs interactively. It should be a drop-in replacement for `srun` for the purposes of this lab, but you can read more details on `salloc` here. Try these commands:

- Running on one node with 4 MPI processes.

  ```
  > salloc --nodes 1 --ntasks 4 mpirun ./hello
  ```

- What if we want to run it on *many* more nodes?

  ```
  > salloc --nodes 4 --ntasks 4 mpirun ./hello
  ```

- What if we want to specify that each node runs 2 tasks (2 MPI processes)?

  ```
  > salloc --nodes 4 --ntasks-per-node 2 mpirun ./hello
  ```

- What if we want to run these tasks on a specific node type?

  ```
  > salloc -p i7-7700 --nodes 1 --ntasks 8 mpirun ./hello
  ```

Exercise 3

Use the last command above to try to find out the maximum number of tasks (MPI processes) that you are allowed to run on a **single node** – try all the different partition types. Does this depend on the number of *physical* or *logical* CPU cores on that node? Try to remove this restriction using `salloc`'s `--overcommit` option.

This co-operation between Slurm and MPI allows for a separation of concerns: MPI programs can focus on communication between each other, and Slurm can handle job scheduling and resource allocation.

Part 3: Mapping MPI Processes to Nodes / Cores

So far, we have let Slurm automatically decide how to map each of our MPI processes to the cluster's nodes and the CPUs on each node. However, we may need more control to get better / more predictable performance. There are many ways to manipulate this mapping (more info in the mpirun documentation), but we will only go through a few here. **This is crucial to achieving good performance with MPI!**

While mapping tasks/processes to nodes, you may encounter an error like this:

```
--------------------------------------------------------------------------
ORTE has lost communication with a remote daemon.

  HNP daemon   : [[12660,0],0] on node soctf-pdc-001
  Remote daemon: [[12660,0],2] on node soctf-pdc-013

This is usually due to either a failure of the TCP network
connection to the node, or possibly an internal failure of
the daemon itself. We cannot recover from this failure, and
therefore will terminate the job.
--------------------------------------------------------------------------
```

This likely means that your mapping was not accepted by `mpirun` and the run failed, not that the network failed as suggested by this error. Try again with different options.

Mapping and Binding

There are two phases of assigning MPI processes to nodes and cores: *mapping*, followed by *binding*.

1. **Mapping**: telling MPI processes which specific node each will execute on, that is, *mapping* processes to specific nodes. The relevant MPI option is `--map-by`. For instance, give nodes A and B allocated by Slurm, `--map-by node` will map the first MPI process to node A, the next one to node B, the next to node A, and so on. There are other options including `map-by core`, `map-by hwthread`, and more.
2. **Binding**: after processes have been mapped to specific nodes, this is the process of assigning each to specific CPUs within the node. The relevant MPI option is `--bind-to`. One possibility is `bind-to none`, which means that MPI processes are free to run on any logical core within the node. Other options include `bind-to core` (each MPI process is bound to one core, and usually both of the hardware threads within that core), `bind-to hwthread` (each MPI process is bound to one specific hardware thread), and `bind-to socket` (each MPI process is bound to one socket).

With these two options (and others), you can achieve precise control over how your MPI program runs, and this heavily impacts your overall performance.

Distributing and Mapping MPI Processes

As the commands in this section have many options for debugging (showing mapping and binding information), we have moved everything into a Slurm batch script called `distribute.sh`. Open this file and read the comments to understand what it does. Now, let's start mapping and binding.

- Run the script on the `hello` executable.

  ```
  > sbatch ./distribute.sh ./hello
  ```

- View the Slurm job log in the `logs/` folder. Observe the debugging information printed by `mpirun`. It should show the nodes that are allocated under "ALLOCATED NODES", the mapping for each node under "JOB MAP", and some information about the binding in the lines starting with "MCW rank ...".
- Since we did not specify `--map-by` and `--bind-to`, we are using the default options – the default behavior can be a little complex. Let's be explicit with our options.
- Let's specify the mapping to be by `node` and the binding to be by `core`.

  ```
  > sbatch ./distribute.sh --map-by node --bind-to core ./hello
  ```

- Notice how even numbered processes are mapped to one node, and odd numbered processes are mapped to another. This is due to our `--map-by` option.
- Notice the binding information for each process – the symbol `B` represents on which logical cores(s) that process can run within the node, each dot represents a logical core (hardware thread), and each "/" symbol represents the separation between two cores in the node.

Exercise 4

Try to achieve these possibilities by changing the mapping and binding options:

1. Keep the same mapping but allow the MPI processes to run on any core.
2. Map and bind each process to separate cores on each of the nodes, where each process should only run on a single hardware thread within each core (i.e., only one hardware thread on each core should be occupied).
3. Try changing to a different partition, changing the number of tasks, and number of nodes in the allocation. Observe the results.

Part 4: Process-to-process Communication

While running completely separate tasks on many machines might be useful, the true power of MPI is in its ability to handle complex communication between each of its processes.

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.

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 5

Open the program `block_comm.cpp` and observe the series of MPI_Send and _Recv calls.

Compile `block_comm.cpp` (refer to Exercise 1 if necessary) and use `salloc` or `sbatch` to run it with *two processes* across *two different nodes*.

Exercise 6

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

Exercise 7

What happens if we flip the order of `MPI_Send` and `MPI_Recv` in the master process? Consider the implications of this.

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 8

Compile the program nblock_comm.cpp and run it with *3 processes* across *3 nodes*. Modify the source (new name nblock_comm_1.cpp) 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/whats-happening/articles/what-mpi-nonblocking-correctness-and-performance

Part 5: For Submission

Exercise 9

View and compile the MPI matrix multiplication program, `mm-mpi.cpp`.

This program implements distributed matrix multiplication. The matrix size must be a power of 2, and the total number of processes must be of the form ($2^n + 1$ for some n).

Your goal in this exercise is to run `mm-mpi` with a matrix size of **2048** using Slurm, across a few configurations, and **explain the differences in performance you see**.

Use these configurations only (but you may add options for debug reasons):

- Run on **one** `i7-7700` node with 5 total processes and include the option `--bind-to hwthread`.
- Run on **one** `xs-4114` node with 5 total processes and include the option `--bind-to hwthread`.
- Run on **one** `i7-7700` **and one** `xs-4114` node with 5 total processes and include the options `--map-by node` and `--bind-to hwthread`. You can do this with `salloc --nodes 2 --ntasks 5 --constraint="[i7-7700*1&xs-4114*1]" mpirun --map-by node --bind-to hwthread ./mm-mpi`

Refer to the communication and computation time reported by the worker processes, and the total time reported by the master process.

Exercise 10

There's an unexpected and significant performance difference these two commands:

- `salloc -n5 -N1 -p i7-7700 mpirun ./mm-mpi 256`
- `salloc -n9 -N1 --overcommit -p i7-7700 mpirun ./mm-mpi 256`

Run and explain the performance gap between these two configurations. You can assume that they both compute the correct result. You may add code such as printf statements for debugging.

Exercise 11

Suggest how could you change `mm-mpi.cpp` to fix the slow performance of the overcommitting configuration, while maintaining correctness. You do not have to submit any code.

Lab sheet (2% of your final grade):
You are required to produce a write-up with the results for exercises 9, 10, and 11. Submit the lab report (in PDF form with file name format A0123456X.pdf (do not zip your submission!)) via Canvas before **Monday, 28 Oct, 2pm**. The document must contain:

- Maximum 1 page.
- Your explanation for exercise 9
- Your explanation for exercise 10
- A brief description of your idea for exercise 11 (you don't have to submit code), and why it works.