Parallel Programming @ NYCU, Fall 2021

This is the webpage for the Parallel Programming course

Programming Assignment IV: MPI Programming

Parallel Programming by Prof. Yi-Ping You

Due date: **23:59, Dec 02, Thursday, 2021**

The purpose of this assignment is to familiarize yourself with MPI programming.

Table of Contents

- Programming Assignment IV: MPI Programming
  - Due Date
  - Setting Up the Environment
  - Part 1: Getting Familiar with MPI Programming
    - 1.1 MPI Hello World, Getting Familiar with the MPI Environment
    - 1.2 Calculating PI with MPI
      - 1.2.1 MPI Blocking Communication & Linear Reduction Algorithm
      - 1.2.2 MPI Blocking Communication & Binary Tree Reduction Communication Algorithm
      - 1.2.3 MPI Non-Blocking Communication & Linear Reduction Algorithm
      - 1.2.4 MPI Collective: MPI_Gather
      - 1.2.5 MPI Collective: MPI_Reduce
      - 1.2.6 MPI Windows and One-Sided Communication & Linear Reduction Algorithm
    - 1.3 Measuring Bandwidth and Latency on NYCU-PP workstations with Ping-Pong
  - Part 2: Matrix Multiplication with MPI
  - Requirements
  - Grading Policy
  - Evaluation Platform
  - Submission
  - References

Get the source code:

```
$ wget https://nycu-sslab.github.io/PP-f21/HW4/HW4.zip
$ unzip HW4.zip -d HW4
$ cd HW4
```

0. Setting Up the Environment

To make MPI work properly, you need to be able to execute jobs on remote nodes without typing a password. You will need to generate an ssh key by yourself. You may follow the instructions below or refer to this article for instruction.

You will also need to configure your `~/.ssh/config` file to define host names. Use `HW4/config` as a template for the configuration. Notice that in the configuration template, the string follows each `User` should be your student ID.

ATTENTION: We will test your code based on this config.

```
# <Login to one of the PP nodes>
$ mkdir -p ~/.ssh
$ ssh-keygen -t rsa # Leave all empty
# <setup the config>
$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

Wait minutes for system setup... Then, login to all machines at least once to make sure the configuration works. This will save the SSH fingerprints of the machines in `~/.ssh/known_hosts`.

```
# Log in without entering password
$ ssh pp1
$ ssh pp2
$ ssh pp3
$ ssh pp4
$ ...
$ ssh pp9
```

1. Part 1: Getting Familiar with MPI Programming

1.1 MPI Hello World, Getting Familiar with the MPI Environment

Here we're going to implement our first MPI program.

Expected knowledge includes basic understanding of the MPI environment, how to compile an MPI program, how to set the number of MPI processes, and retrieve the rank of the process and number of MPI processes at runtime.

Let's learn how to launch an MPI code on NYCU-PP workstations.

There is a starter code for you. Look at `hello.cc` . Please finish the _TODO_s with the `MPI_Comm_size` and `MPI_Comm_rank` functions.

We will work on NFS (which is already set up), so we don't need to copy and compile the code again and again on different nodes.

Let's first compile `hello.cc` on NFS:

```
$ cd ./HW4
$ mpicxx ./hello.cc -o mpi_hello
```

Then create a file `hosts` , which is a text file with hosts specified, one per line. We will use the file to specify on which hosts to launch MPI processes when running `mpirun` . For example, we put the following two lines into the file to specify we will launch MPI processes on nodes `pp1` and `pp2` . Notice that you should use the same hostnames as in `~/.ssh/config` .

`hosts` :

```
pp1 # 1st node hostname
pp2 # 2ed node hostname
```

Make sure `mpi_hello` is located in the same directory hierarchy on the two nodes (should not be a problem under NFS) before running `mpirun` to start MPI processes:

```
$ mpirun -np 8 --hostfile hosts mpi_hello
```

This command will launch eight processes to run `mpi_hello` on the two nodes, and you should be able to get an output similar to:

```
Hello world from processor ec037-072, rank 0 out of 8 processors
Hello world from processor ec037-072, rank 1 out of 8 processors
Hello world from processor ec037-072, rank 3 out of 8 processors
Hello world from processor ec037-072, rank 2 out of 8 processors
Hello world from processor ec037-073, rank 4 out of 8 processors
Hello world from processor ec037-073, rank 6 out of 8 processors
Hello world from processor ec037-073, rank 5 out of 8 processors
Hello world from processor ec037-073, rank 7 out of 8 processors
```

PS: There may be some `No protocol specified` messages in the beginning, it's ok to just ignore them.

Code Implementation: (*3 points*)

- Complete `hello.cc` : the Hello World code in C.
- `mpirun -np 8 --hostfile hosts mpi_hello` should run correctly
- The argument of `-np` is expected to be any positive integer less than or equal to 16.

Q1 (*3 points*)

1. How do you control the number of MPI processes on each node? (*1.5 points*)
2. Which functions do you use for retrieving the rank of an MPI process and the total number of processes? (*1.5 points*)

1.2 Calculating PI with MPI

In this exercise, we are going to parallelize the calculation of Pi following a Monte Carlo method and using different communication functions and measure their performance.

Expected knowledge is MPI blocking and non-blocking communication, collective operations, and one-sided communication.

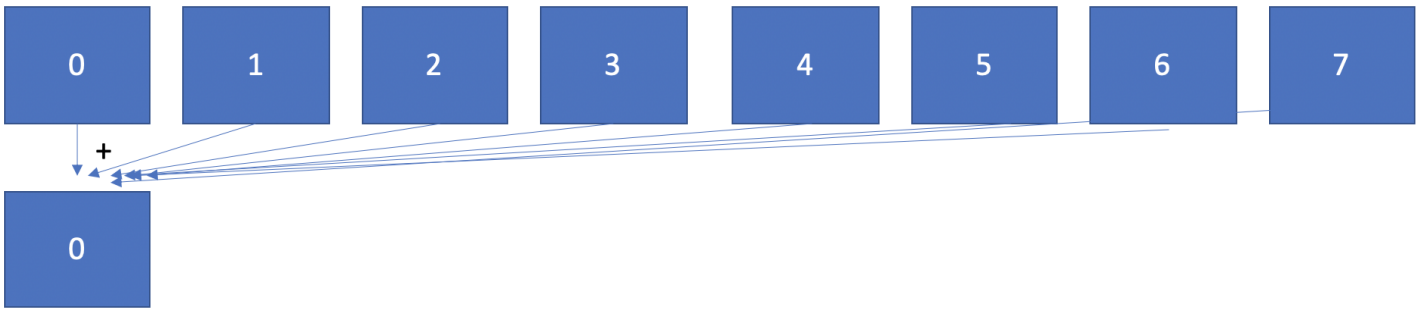
Instructions: Write different versions of MPI codes to calculate Pi. The technique we are implementing relies on random sampling to obtain numerical results. You have already implemented this in the previous assignments.

1.2.1 MPI Blocking Communication & Linear Reduction Algorithm

In our parallel implementation, we split the number of iterations of the main loop into all the processes (i.e., `NUM_ITER / num_ranks`). Each process will calculate an intermediate `count` , which is going to be used afterward to calculate the final value of Pi.

To calculate Pi, you need to send all the intermediate counts of each process to rank 0. This communication algorithm for reduction operation is called linear as the communication costs scales as the number of processes.

An example of linear reduction with eight processes is as follows:



Rank 0 is going to receive the intermediate `count` calculated by each process and accumulate them to its own count value. (Rank 0 also needs to calculate a `count`, and this stands for the following sections in Part One.) Finally, after rank 0 has received and accumulated all the intermediate counts, it will calculate Pi and show the result, as in the original code.

Implement this code using blocking communication and test its performance.

> *Hint* 1. Change the main loop to include the number of iterations per process, and not `NUM_ITER` (which is the total number of iterations).
>
> *Hint* 2. Do not forget to multiply the seed of `srand()` with the rank of each process (e.g., "rank * SEED") to ensure the RNGs of processes generate different random numbers.

There is a starter code `pi_block_linear.cc` for you.

```
$ mpicxx pi_block_linear.cc -o pi_block_linear; mpirun -np 4 --hostfile hosts pi_block_linear 1000000000
```

Code Implementation: (*7 points*)

- Complete `pi_block_linear.cc` : the MPI block communication for PI.
- Implement the code using a **linear algorithm** with `MPI_Send` / `MPI_Recv` .
- Do not modify the output messages.
- `$ mpicxx pi_block_linear.cc -o pi_block_linear; mpirun -np 4 --hostfile hosts pi_block_linear 1000000000` should run correctly.
- The argument of `-np` is expected to be any positive integer less than or equal to 16.
- `mpirun -np 4 --hostfile hosts pi_block_linear 1000000000` should run almost as fast as TA's reference version (refer to `/HW4/ref`). The difference (in execution time) should be within 15%.

> You may use up to five nodes (and up to four processors in each node) for Part 1, so you may want to try different configurations in the hostfile.

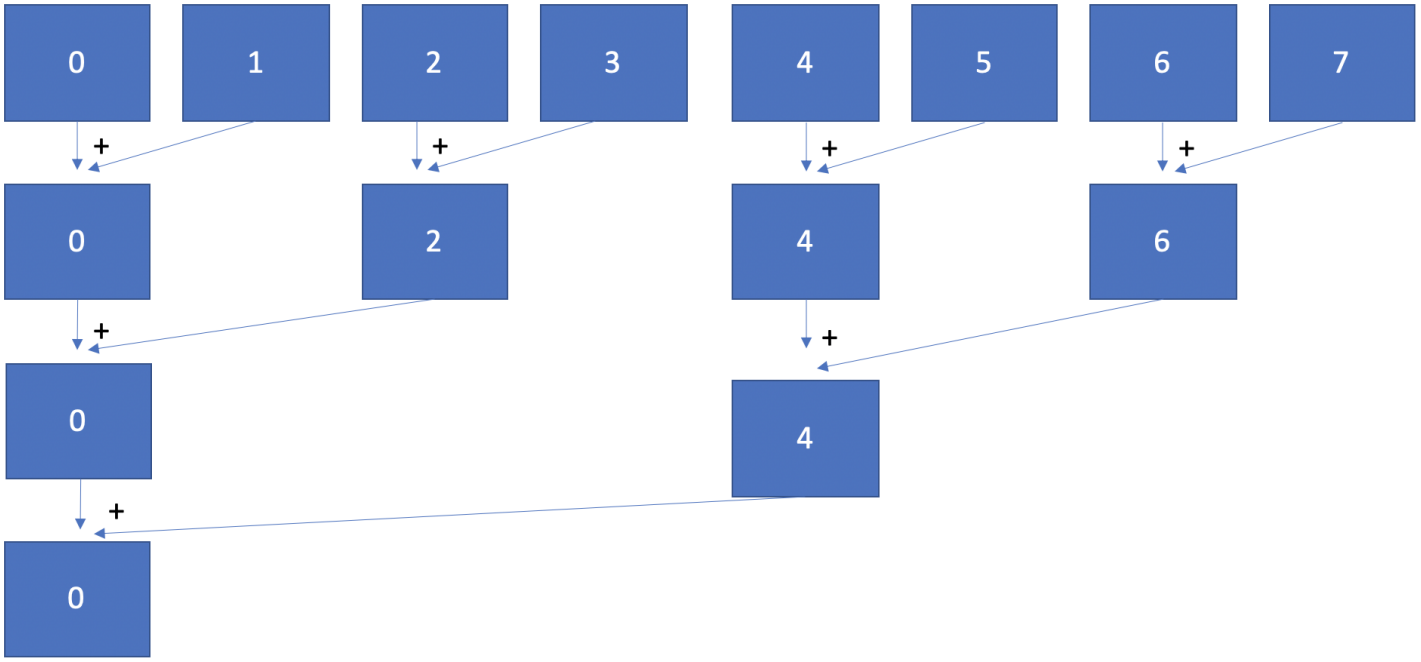
Q2 (*2 points*)

1. Why `MPI_Send` and `MPI_Recv` are called "blocking" communication? (*1 points*)
2. Measure the performance (execution time) of the code for 2, 4, 8, 12, 16 MPI processes and plot it. (*1 points*)

1.2.2 MPI Blocking Communication & Binary Tree Reduction Communication Algorithm

Implement the binary tree communication algorithm for performing the reduction on rank 0 using blocking communication (e.g., `MPI_Send` / `MPI_Recv`).

The communication pattern for a reduction with a binary tree with eight processes is as follows:



In you implementation, you can assume that we use a power-of-two number of processes.

There is a starter code `pi_block_tree.cc` for you.

```
$ mpicxx pi_block_tree.cc -o pi_block_tree; mpirun -np 4 --hostfile hosts pi_block_tree 1000000000
```

Code Implementation: (*7 points*)

- Complete `pi_block_tree.cc` : the MPI block communication for PI.
- Implement the code using a **binary tree reduction algorithm** with `MPI_Send` / `MPI_Recv` .
- Do not modify the output messages.
- `$ mpicxx pi_block_tree.cc -o pi_block_tree; mpirun -np 4 --hostfile hosts pi_block_tree 1000000000` should run correctly.
- The argument of `-np` is expected to be any power-of-two number less than or equal to 16.
- `mpirun -np 4 --hostfile hosts pi_block_tree 1000000000` should run almost as fast as TA's reference version. The difference (in execution time) should be within 15%.

Q3 (*6 points*)

1. Measure the performance (execution time) of the code for 2, 4, 8, 16 MPI processes and plot it. (*1 points*)
2. How does the performance of binary tree reduction compare to the performance of linear reduction? (*2 points*)
3. Increasing the number of processes, which approach (linear/tree) is going to perform better? Why? Think about the number of messages and their costs. (*3 points*)

1.2.3 MPI Non-Blocking Communication & Linear Reduction Algorithm

Use non-blocking communication for the linear reduction operation (in Section 1.2.1).

> *Hint*: Use a non-blocking `MPI_Irecv()` (MPI Receive with Immediate return). The basic idea is that rank 0 is going to issue all the receive operations and then wait for them to finish. You can either use `MPI_Wait()` individually to wait for each request to finish or `MPI_Waitall()` . Regardless of your decision, keep in mind that we want you to perform the receive operations in parallel. Thus, do not call `MPI_Irecv()` and immediately `MPI_Wait()` ! In addition, we recommend you allocate an array of `MPI_Request` and also an array of counts (i.e., one for each receive needed).

There is a starter code `pi_nonblock_linear.cc` for you.

```
$ mpicxx pi_nonblock_linear.cc -o pi_nonblock_linear; mpirun -np 4 --hostfile hosts pi_nonblock_linear 1000000000
```

Code Implementation: (*7 points*)

- Complete `pi_nonblock_linear.cc` : the MPI non-blocking communication for PI.
- Implement the code using a **non-blocking algorithm** with `MPI_Send` / `MPI_IRecv` / `MPI_Wait` / `MPI_Waitall` .
- Do not modify the output messages.
- `$ mpicxx pi_nonblock_linear.cc -o pi_nonblock_linear; mpirun -np 4 --hostfile hosts pi_nonblock_linear 1000000000` should run correctly.
- The argument of `-np` is expected to be any positive integer less than or equal to 16.
- `mpirun -np 4 --hostfile hosts pi_nonblock_linear 1000000000` should run almost as fast as TA's reference version. The difference (in execution time) should be within 15%.

Q4 (*5 points*)

1. Measure the performance (execution time) of the code for 2, 4, 8, 12, 16 MPI processes and plot it. (*1 points*)
2. What are the MPI functions for non-blocking communication? (*1 points*)
3. How the performance of non-blocking communication compares to the performance of blocking communication? (*3 points*)

1.2.4 MPI Collective: MPI_Gather

Use the collective `MPI_Gather()` operation, instead of point-to-point communication.

Hint: You can keep rank 0 as the root of the communication and still make this process aggregate manually the intermediate counts. Remember that the goal of `MPI_Gather()` is to provide the root with an array of all the intermediate values. Reuse the array of counts as the output for the gather operation.

There is a starter code `pi_gather.cc` for you.

```
$ mpicxx pi_gather.cc -o pi_gather; mpirun -np 4 --hostfile hosts pi_gather 1000000000
```

Code Implementation: (*7 points*)

- Complete `pi_gather.cc` : the MPI gather communication for PI.
- Implement the code using `MPI_Gather` .
- Do not modify the output messages.
- `$ mpicxx pi_gather.cc -o pi_gather; mpirun -np 4 --hostfile hosts pi_gather 1000000000` should run correctly.
- The argument of `-np` is expected to be any positive integer less than or equal to 16.
- `mpirun -np 4 --hostfile hosts pi_gather 1000000000` should run almost as fast as TA's reference version. The difference (in execution time) should be within 15%.

Q5 (*1 points*)

1. Measure the performance (execution time) of the code for 2, 4, 8, 12, 16 MPI processes and plot it. (*1 points*)

1.2.5 MPI Collective: MPI_Reduce

Use the collective `MPI_Reduce()` operation.

> *Hint* 1: Remember that the goal of `MPI_Reduce()` is to perform a collective computation. Use the `MPI_SUM` operator to aggregate all the intermediate count values into rank 0, But, watch out: rank 0 has to provide its own count as well, alongside the one from the other processes.
>
> *Hint* 2: The send buffer of `MPI_Reduce()` must not match the receive buffer. In other words, use a different variable on rank 0 to store the result.

There is a starter code `pi_reduce.cc` for you.

```
$ mpicxx pi_reduce.cc -o pi_reduce; mpirun -np 4 --hostfile hosts pi_reduce 1000000000
```

Code Implementation: (*7 points*)

- Complete `pi_reduce.cc` : the MPI reduce communication for PI.
- Implement the code using `MPI_Reduce` .
- Do not modify the output messages.
- `$ mpicxx pi_reduce.cc -o pi_reduce; mpirun -np 4 --hostfile hosts pi_reduce 1000000000` should run correctly.
- The argument of `-np` is expected to be any positive integer less than or equal to 16.
- `mpirun -np 4 --hostfile hosts pi_reduce 1000000000` should run almost as fast as TA's reference version. The difference (in execution time) should be within 15%.

Q6 (*1 points*)

1. Measure the performance (execution time) of the code for 2, 4, 8, 12, 16 MPI processes and plot it. (*1 points*)

1.2.6 MPI Windows and One-Sided Communication & Linear Reduction Algorithm

Use MPI Windows and MPI one-sided communication, which we didn't cover this in class. You can choose the functions you'd like to use but remember that there is a reduction on the same MPI window from many processes! You may refer to `one_side_example.c` to get familiar with it.

There is a starter code `pi_one_side.cc` for you.

```
$ mpicxx pi_one_side.cc -o pi_one_side; mpirun -np 4 --hostfile hosts pi_one_side 1000000000
```

Code Implementation: (*7 points*)

- Complete `pi_one_side.cc` : the MPI one sided communication for PI.
- Implement the code using `MPI_Widows_*` (or `MPI_Put` , `MPI_Get` , and `MPI_Accumulate`). You only need to implement the first one, and the remainings are optional.
- Do not modify the output messages.
- `$ mpicxx pi_one_side.cc -o pi_one_side; mpirun -np 4 --hostfile hosts pi_one_side 1000000000` should run correctly.
- The argument of `-np` is expected to be any positive integer less than or equal to 16.
- `mpirun -np 4 --hostfile hosts pi_one_side 1000000000` should run almost as fast as TA's reference version. The difference (in execution time) should be within 15%.

Q7 (*5 points*)

1. Measure the performance (execution time) of the code for 2, 4, 8, 12, 16 MPI processes and plot it. (*1 points*)
2. Which approach gives the best performance among the **1.2.1**-**1.2.6** cases? What is the reason for that? (*4 points*)

1.3 Measuring Bandwidth and Latency on NYCU-PP workstations with Ping-Pong

Expected knowledge is concepts of bandwidth and latency and performance model for parallel communication.

The ping-pong is a benchmark code to measure the bandwidth and latency of a supercomputer. In this benchmark, two MPI processes use `MPI_Send` and `MPI_Recv` to continually bounce messages off of each other until a final limit.

The ping-pong benchmark provides as output the average time for the ping-pong for different messages sizes.

If we plot the results of the ping-pong with size on the x-axis and ping-pong time on the y-axis, we will roughly obtain points distributed along a line. If we do a linear best fit of the obtained points, we will get the intercept and the slope of the line. The inverse of the line slope is the bandwidth while the intercept is the latency of the system.

We will use `ping_pong.c` to measure the ping-pong time for different message sizes.

Instructions. For completing this exercise, you may refer to this video: Performance Modeling & Ping-Pong

Run the ping-pong code and calculate the bandwidth and latency for:

- case 1: intra-node communication (two processes on the same node), and
- case 2: inter-node communication (two processes on different nodes). You should consider more different nodes in this case.

Q8 (*9 points*)

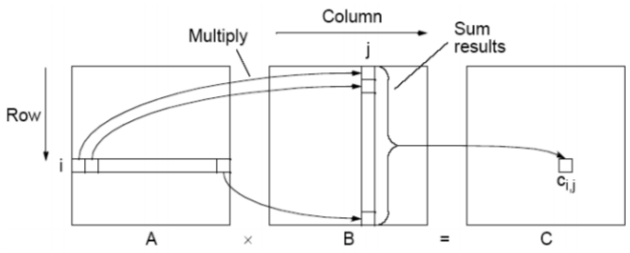
1. Plot ping-pong time in function of the message size for cases 1 and 2, respectively. (*4 points*)
2. Calculate the bandwidth and latency for cases 1 and 2, respectively. (*5 points*)

2. Part 2: Matrix Multiplication with MPI

Write a MPI program which reads an $n \times m$ matrix A and an $m \times l$ matrix B, and prints their product, which is stored in an $n \times l$ matrix C. An element of matrix C is obtained by the following formula:

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$$

where a_{ij}, b_{ij} and c_{ij} are elements of A, B, and C, respectively.



Your mission is to calculate the matrix multiplication as fast as possible. You may try many methods (Ex: tiling or some algorithms. Note that SIMD isn't allowed here!) to achieve a higher performance. You are allowed to use any code on the Internet, but you need to re-write the code yourself, which means copying and pasting is not allowed. In addition, you are not allowed to use any third-party library; that is, you have to write your MPI program from scratch. Furthermore, each node is allowed to run a single-threaded process, so using OpenMP, Pthread, fork, GPU, etc. are not allowed.

> If you are not sure about the rule, ask on our Facebook group!

Input

In the first line, three integers n, m and l are given separated by space characters.

In the following lines, the $n \times m$ matrix A and the $m \times l$ matrix B are given. All the numbers are integers.

Output

Print elements of the $n \times l$ matrix C (c_{ij}). Print a single space character after "each" element!

Constraints

1. $1 \leq n, m, l \leq 10000$
2. $0 \leq a_{ij}, b_{ij} \leq 100$

Sample Input

```
3 2 3
1 2
0 3
4 5
1 2 1
0 3 2
```

Sample Output

```
1 8 5
0 9 6
4 23 14
```

Note: There is " \n" at the end of each line!

Data Sets

There are two data sets. The table shows the credit, the range of n, m, and l, and the hosts used for each data set.

Set	Credit	n, m, l	Hosts
1	*10 points*	300-500	Any of four from `pp1` - `pp9`
2	*10 points*	1000-2000	`pp1` - `pp9`

You may have different implementations for the two data sets.

Notice that only one process per node is allowed.

Here is an example hostfile:

```
pp1 slots=1
pp3 slots=1
pp5 slots=1
pp7 slots=1
```

Q9 (*5 points*)

1. Describe what approach(es) were used in your MPI matrix multiplication for each data set.

3. Requirements

3.1 Requirements for Part One

- Code implementation (following the instructions in Sections 1.2.1-1.2.6
  - `hello.cc`
  - `pi_block_linear.cc`
  - `pi_block_tree.cc`
  - `pi_gather.cc`
  - `pi_nonblock_linear.cc`
  - `pi_one_side.cc`
  - `pi_reduce.cc`
- Answer the questions (marked with "**Q1-Q8**") in a **REPORT** using HackMD. (Notice that in this assignment a higher standard will be applied when grading the quality of your report.)

3.2 Requirements for Part Two

- Code implementation
  - You should write a program that fits the input/output criteria as described in Section 2.
  - You should write a `Makefile` that aims to generate an executable called `matmul`. In addition, `main.cc`, which contains the `main()` function, must be a part of `matmul` and cannot be modified.
  - Your program will be evaluated with command `$ make; mpirun -np %N% --hostfile %hosts% matmul < %data_file%`, where `N` is 2-8, `hosts` is our hostfile, and `data_file` is the input.
  - There are some data in the `/HW4/data` folder for testing. The data are available at `/HW4/data.zip` if you'd like to test them in your own machine(s). The testing script provided uses the data to verify your program. You don't expect to pass only the testing data.
- Answer the questions (marked with "**Q9**") in a **REPORT** using HackMD. (Notice that in this assignment a higher standard will be applied when grading the quality of your report.)

4. Grading Policy

NO CHEATING!! You will receive no credit if you are found cheating.

Total of 102%:

- part 1 (77%):
  - Implementation correctness and performance: 45%
  - Questions: 32%
- part 2 (25%):
  - Performance: 20% (10% for each data set)
    - Evaluations will be conducted for each data set with the metric below.
  - Questions: 5%

Metric (for each data set):

$$\frac{T-Y}{T-F}\times 60\%, \text{if } Y < T + \begin{cases} 40\%, \text{if } Y < F \times 2 \\ 20\%, \text{else} \end{cases}$$

where Y and F indicate the execution time of your program and the fastest program, respectively, and $T = F \times 1.5$.

5. Evaluation Platform

Your program should be able to run on UNIX-like OS platforms. We will evaluate your programs on the workstations dedicated for this course. You can access these workstations by `ssh` with the following information.

The workstations are based on Ubuntu 20.04 with Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz processors. `g++-10`, `clang++-11`, and `mpi4` have been installed.

IP	Port	User Name	Password
140.113.215.195	37072 ~ 37080	{student_id}	{your passwd}

Login example:

```
$ ssh <student_id>@140.113.215.195 -p <port>
```

You can use the testing script `test_hw4` to check your answer *for reference only*. Run `test_hw4` in a directory that contains your `HW4_XXXXXXX.zip` file on the workstation.

`test_hw4` :

- Check if the zip file is correct.
- Part 1: Check the correctness. Not check the performance.
- Part 2: Evaluate the performance for the public data (in average time).
- Check the `url.txt`

6. Submission

All your files should be organized in the following hierarchy and zipped into a `.zip` file, named `HW4_xxxxxxx.zip` , where `xxxxxxx` is your student ID.

Directory structure inside the zipped file:

- `HW4_xxxxxxx.zip` (root)
  - `part1`
    - `hello.cc`
    - `pi_block_linear.cc`
    - `pi_block_tree.cc`
    - `pi_gather.cc`
    - `pi_nonblock_linear.cc`
    - `pi_one_side.cc`
    - `pi_reduce.cc`
  - `part2`
    - `Makefile`
    - `main.cc`
    - Other program files you added
  - `url.txt`

Notice that you just need to provide the URL of your HackMD report in `url.txt` , and enable the write permission for someone who knows the URL so that TAs can give you feedback directly in your report.

Zip the file:

```
$ zip HW4_xxxxxxx.zip -r part1 part2 url.txt
```

Be sure to upload your zipped file to new E3 e-Campus system by the due date.

You will get *NO POINT* if your ZIP's name is wrong or the ZIP hierarchy is incorrect.

7. References

- Open MPI 4.0 Document
- MPI Tutorial
- William Gropp, One-sided Communication in MPI