FIT3143 Lab #5 (Week 10)

MESSAGE PASSING INTERFACE (MPI) III

OBJECTIVES

- Explore MPI Virtual Topologies, Master/Slave Design Paradigm, and Time Synchronization.

MARKS

- The lab session is worth **8 marks** of the unit's final mark.

LAB INSTRUCTIONS

1. Preparation is recommended and required for this Lab session. Please **DO NOT** plan to complete the Lab without any preparation/understanding.
2. Students must submit their answers and codes via Moodle in the required format (report in PDF and codes in C/C++ files) before the end of the allocated class. A **late penalty** (5% per day) will apply if you do not submit the required materials on time!
3. The purpose of these biweekly Lab sessions is for students to get an opportunity to test their understanding of the lecture/pre-reading material with a low cost in lost marks if they have not understood the material, rather than not understanding the material.
4. Students should start working on the Lab tasks prior to starting the class.
5. Students' (or Team's) codes and answers will be reviewed and assessed in a coding presentation, peer review, or question-and-answer format.
6. For **team/group presentations**, marks will be allocated for:
    - The correctness of the codes or results
    - The quality and clarity of the presentation
    - The correctness of the explanation during the presentation

    The presentation content can include slides or a format that is suitable. Optionally, marks could be awarded to students capable of asking quality and insightful questions about their peers' presentations. Students can also opt to demonstrate the code, results and observations using a document in lieu of a presentation slide. Nevertheless, the quality/correctness also applies to a demonstrated work, submitted code and documentation containing the results, analysis and observations.
7. For **individual interviews/Q&A sessions**, marks will be allocated for:

    - The correctness of the verbal answer
    - The quality of the verbal answer

   Try to focus on the most important and critical idea in the answer. Marks will be awarded on the student's ability to explain their ideas with reference to the submitted code or documents in a concise, focused and accurate manner. Answers with errors, inaccurate or lengthy explanations will lose marks. Showing little to no understanding of the code, associated libraries and submitted results/analysis will also result in losing marks.
8. Always make sure to read the marking rubric.
9. Marks will not be awarded if you skip the class, or do not make any submissions, or do not make any contributions in the working group/team, or make an empty submission to Moodle (unless a special consideration extension has been approved).
10. You are allowed to use Generative-AI to search for information and resources during the pre-class and in-class preparation period. However, you must declare in your report and upload all the prompt records (in a PDF file).
11. AI tools are not allowed during the presentation period or any oral/coding interview sessions.
12. Certain tasks or activities may be performed in a group/team format. However, students must still submit all the tasks (required files) individually in Moodle.
13. All submitted files should include students' names, ID and Monash email addresses.

LAB ACTIVITIES

Students are required to form teams as advised by the teaching staff. At the Clayton campus, each team should consist of approximately 4 to 6 students, while at the Malaysia campus, each team should consist of 2 to 4 students.

❖ Preparation period: You will have a **maximum of 40 minutes** (per team) to work on the Tasks during the session. You should work with your team members to complete the tasks before the start of the session, in particular any tasks labelled with [Warm-up Task]. Private coding repositories (e.g. GitHub) are recommended to share and work with your team members.

❖ Presentation Period: For each team, you are required to present or demonstrate your code, results and observations during the lab session. Additionally, documents or slides containing an explanation of the code design, results and analysis should be submitted as part of the presentation or lab demonstration. If a team is not comfortable presenting the work in person during the lab, the team can opt to record and submit video file(s) to Moodle prior to the lab session. The lab tutor will play the video in the class and review it during the lab session. However, all team members should be present in the lab session for the Q&A period with the lab tutor.

❖ Q&A Period: Each team member will be asked one to two questions based on the submitted and presented work.

❖ The overall session per team should be **between 10 and 15 minutes** long, depending on the size of the team.

Submission Checklist

- Core Task - All coding and supporting files.
- Extended Task - All coding and supporting files.
- Warm-Up Task **[Optional]** – Coding and supporting files are optional.
- Documentation describing the code design, results and analysis or observations (where applicable).
    - The documentation can be in the format of slides, docx, PDF or an equivalent format, but choose only one format.
    - Although the marking rubric requires the submitted content to show a deep understanding of the topics with materials, please aim to keep the length of the slides or documentation to within a 10 to 15-minute presentation duration.
    - Focus on emphasising the design, results and observations that would demonstrate

a good understanding of the lab tasks.
- A separate AI declaration (in PDF format), **if not** already declared in the submitted documentation as aforementioned.

LAB ACTIVITIES

Warm-Up Task – Getting used to manual pages [Not assessed]

Open the manual pages https://docs.open-mpi.org/en/v5.0.x/man-openmpi/index.html

Make sure it sits comfortably in your browser, and you can quickly access the API descriptions as quickly as you can!

Warm-Up Task – Creating a 2D Cartesian grid with OpenMPI [Not assessed]

	0	**1**	**2**	**3**	**4**
0	Rank0 (0,0)	Rank1 (0,1)	Rank2 (0,2)	Rank3 (0,3)	Rank4 (0,4)
1	Rank5 (1,0)	Rank6 (1,1)	Rank7 (1,2)	Rank8 (1,3)	Rank9 (1,4)
2	Rank10 (2,0)	Rank11 (2,1)	Rank12 (2,2)	Rank13 (2,3)	Rank14 (2,4)
3	Rank15 (3,0)	Rank16 (3,1)	Rank17 (3,2)	Rank18 (3,3)	Rank19 (3,4)

Figure 1: Cartesian grid layout

In this task, you are required to create a 2D grid using MPI Cartesian topology functions. Figure 1 illustrates an example of a 4 by 5 Cartesian grid. For this task, each MPI process (in the grid) is required to print the following:

a) Current rank
b) Cartesian rank
c) Coordinates
d) List of immediate adjacent processes (left, right, top and bottom).

Users must also have an option to specify the grid size by command line argument(s). Sample code has been provided for you. You are required to fill in 5 lines of code in highlighted areas (in red).

[Hint: Look for Open MPI functions starting with “MPI_Cart”.]

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <mpi.h>

#define SHIFT_ROW 0
#define SHIFT_COL 1
#define DISP 1

int main(int argc, char *argv[]) {

     int ndims=2, size, my_rank, reorder, my_cart_rank, ierr;
     int nrows, ncols;
     int nbr_i_lo, nbr_i_hi;
     int nbr_j_lo, nbr_j_hi;
     MPI_Comm comm2D;
     int dims[ndims],coord[ndims];
     int wrap_around[ndims];

     /* start up initial MPI environment */
     MPI_Init(&argc, &argv);
     MPI_Comm_size(MPI_COMM_WORLD, &size);
     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

     /* process command line arguments*/
     if (argc == 3) {
          nrows = atoi (argv[1]);
          ncols = atoi (argv[2]);
          dims[0] = nrows; /* number of rows */
          dims[1] = ncols; /* number of columns */
          if( (nrows*ncols) != size) {
               if( my_rank == 0) printf("ERROR: nrows*ncols)=%d *
%d = %d != %d\n", nrows, ncols, nrows*ncols,size);
               MPI_Finalize();
               return 0;
          }
     } else {
          nrows=ncols=(int)sqrt(size);
          dims[0]=dims[1]=0;
     }
```

```
/************************************************************
*/
/* create cartesian topology for processes */
/************************************************************
*/
     MPI_Dims_create(size, ndims, dims);
     if(my_rank==0)
          printf("Root Rank: %d. Comm Size: %d: Grid Dimension =
[%d x %d] \n",my_rank,size,dims[0],dims[1]);

     /* create cartesian mapping */
     wrap_around[0] = wrap_around[1] = 0; /* periodic shift is
.false. */
     reorder = 1;
     ierr =0;
     /* ierr = ... */

     if(ierr != 0) printf("ERROR[%d] creating CART\n",ierr);

     /* find my coordinates in the cartesian communicator group */
     /* . . .*/
     /* use my cartesian coordinates to find my rank in cartesian
group*/
     /* . . . */
     /* get my neighbors; axis is coordinate dimension of shift */
     /* axis=0 ==> shift along the rows: P[my_row-1]: P[me] :
P[my_row+1] */
     /* axis=1 ==> shift along the columns P[my_col-1]: P[me] :
P[my_col+1] */

     /* . . . */
     /* . . . */

     printf("Global rank: %d. Cart rank: %d. Coord: (%d, %d).
 Left: %d. Right: %d. Top: %d. Bottom: %d\n", my_rank,
 my_cart_rank, coord[0], coord[1], nbr_j_lo, nbr_j_hi, nbr_i_lo,
 nbr_i_hi);
     fflush(stdout);

     MPI_Comm_free( &comm2D );
     MPI_Finalize();
     return 0;
}
```

Warm-Up Task – IPC between adjacent processes in a Cartesian grid [Not assessed]

Based on the previous task, implement the following:

1. Each MPI process in the grid will generate a *random prime number* and exchange the generated value with *all of* its adjacent processes.
2. Upon exchanging the random numbers, each process then compares the received prime numbers with its own prime number.
3. For any received prime numbers from adjacent processes, if there is a match between the received prime numbers with the process's own prime number, the process will log a record in a text file. The record must contain the prime number being matched, the rank of the receiving process, and the rank of the sending process.

   Note: If you are simulating a 4 × 5 grid, we expect 20 unique log files. We assume if the log file is empty (or missing), then there are no prime numbers being matched for the receiving process. For ease of marking, please name the log file based on the rank of the receiving process (e.g., rank_0.txt).
4. Implement your program looping part a) to c) for at least 500 (or more) times. Make sure your program can generate at least a few records of matching prime numbers.
5. How can we further optimise the basic program to *reduce overhead/improve networking performance*? For this task, you are required to implement an optimised version by further refining the basic version.

Warm-Up Task - Master/Slaves architecture with Open MPI [Not assessed]

Message passing is well-suited to handling computations where a task is divided up into subtasks, with most of the processes used to compute the subtasks and a control/master process (or a small subset of control/master processes) managing the tasks.

The manager is often called the "master" and the other processes the "workers"/"slaves".

In this task, you will learn how to build an Input/Output master/slave system. This will allow you to easily arrange different kinds of input and output from your program, including:

- Ordered output (results from process 2 must be printed after process 1)
- Duplicate removal (a single instance of "Hello world" instead of one from each process)
- Input to all processes from a terminal (not asking for a user to input n times for n processes)

This will be accomplished by dividing the processes (in MPI_COMM_WORLD) into two sets — the master and the slaves. In general, the master process is designed for input operations, output operations, workload partitioning/distribution, and process coordination. For the slaves, they are mainly served for running/solving the partitioned/distributed tasks.

In this task, you will need to divide the processors and split the MPI_COMM_WORLD into two communicators. One communicator will be covering the master and the other communicator covering the slaves. The master will accept messages from the slaves (of type MPI_CHAR) and print them in rank ascending order (that is, slave 0 will be printed before slave 1). Each of the

slaves will send 2 messages to the master. For example, slave 3 will send:
Message 1:

```
Hello from slave 3
```

Message 2:

```
Goodbye from slave 3
```

For simplicity, you may assume the maximum length of all the messages in this task is fixed (e.g., 256 characters) in C.

For this task, you are not allowed to use `intercommunicators`. You will also observe how we use the new communicator for the slaves. Sample code has been provided for you. You are required to fill in 3 lines of code in highlighted areas (in red).

Sample solution:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>

int master_io(MPI_Comm master_comm, MPI_Comm comm);
int slave_io(MPI_Comm master_comm, MPI_Comm comm);

int main(int argc, char **argv)
{
int rank;
MPI_Comm new_comm;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
/* Using MPI_Comm_split to create a new communicator (new_comm)*/
if (rank == 0)
master_io( MPI_COMM_WORLD, new_comm );
else
slave_io( MPI_COMM_WORLD, new_comm );
MPI_Finalize();
return 0;
}

/* This is the master */
int master_io(MPI_Comm master_comm, MPI_Comm comm)
{
int i,j, size;
char buf[256];
MPI_Status status;
MPI_Comm_size( master_comm, &size );
for (j=1; j<=2; j++) {
for (i=1; i<size; i++) {
```

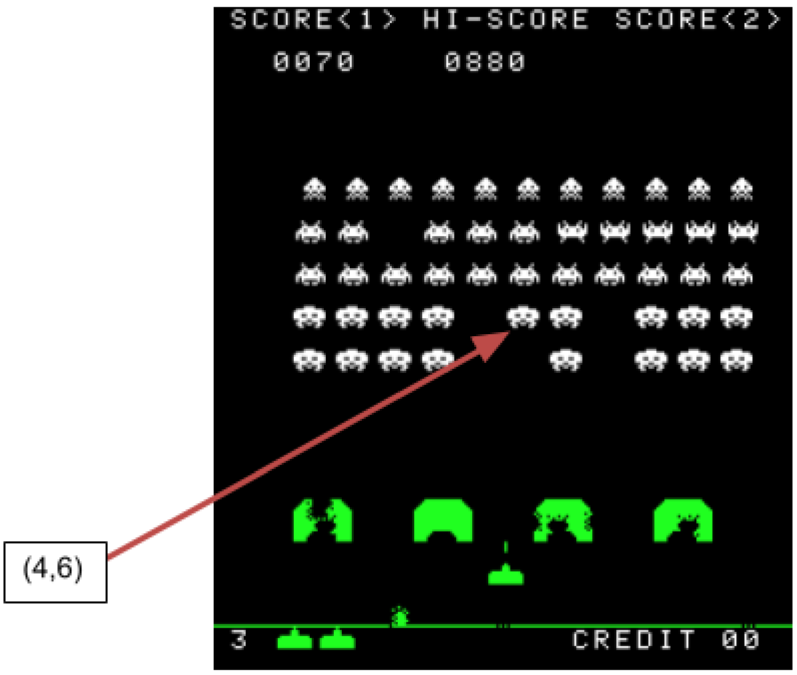
```
MPI_Recv( buf, 256, MPI_CHAR, i, 0, master_comm,&status);
fputs( buf, stdout );
}
}
return 0;
}

/* This is the slave */
int slave_io(MPI_Comm master_comm, MPI_Comm comm)
{

char buf[256];
int rank;

MPI_Comm_rank(comm, &rank);
sprintf(buf, "Hello from slave %d\n", rank);
/*Sending data in buffer to master*/

sprintf( buf, "Goodbye from slave %d\n", rank );
/*Sending data in buffer to master*/
return 0;
}
```

Core Task – Simulation for Space Invaders

In the core task, you are required to help a gaming company implement a simulation prototype for a variant of space invaders. The gaming company has the following specific requirements:

a) The company has abundant machines sitting in a datacenter connected by a low-latency, high-bandwidth network, but each machine has only limited processors (cores).

b) The alien invaders (enemies) are distributed and positioned in ***n*** rows and ***m*** columns, where ***n*** and ***m*** will change depending on the progress/gaming levels. The player controls the cannon located at the bottom, and the goal of the player is to control the cannon to kill all the alien invaders. Below shows an example image of the famous Space Invaders.



https://upload.wikimedia.org/wikipedia/en/2/20/SpaceInvaders-Gameplay.gif

c) For simplicity, we assume the (player) cannon can only fire cannon balls to the bottom-most (surviving) invader in the same column of the cannon (e.g., invader (4,6)). Similarly, the invaders can only shoot cannonballs down in the same column. For each column, we also assume only the bottom-most (surviving) invader can shoot cannonballs. We also ignore the implementation of bunkers (the four green objects between the cannon and the invaders).

d) The gaming company wants to implement a simulation prototype to test the feasibility and efficiency of the game running in the datacenter, with one MPI process executing the game logic of each invader. Additionally, the code should also have 1 MPI process handling the player's game logic (including the player's cannon). General coordination tasks, input & output operations, workload partitioning/distribution, and process coordination should also be implemented in the player MPI process.

e) The gaming logic for the simulation prototype is described as follows:

   i) At the initial time point, the player's cannon starts in the first column (on the left). The player can choose to move left or move right, or stay in the same column for every 1 second. If the player wants to move the cannon beyond the bounds (e.g., one column left from the leftmost column or one column right from the rightmost column), it will be treated as staying in the same column.

ii) A cannonball from the player's cannon will be fired automatically every 1 second, i.e., fired immediately after moving or deciding not to move.

iii) The cannon ball will take 2 seconds to travel to the invaders in the bottom row, and 1 extra second for each additional row (i.e., n+1 seconds for the n^{th} row counting from bottom).

iv) Alien invaders are not allowed to move, and they will be killed immediately when getting shot by the player's cannonball. A cannonball may be fired from each alien invader every 4 seconds, with a (uniform) probability of 0.1 (i.e., 10% chance of firing for every 4 seconds).

v) Similar to the player's cannon ball, the invaders' cannon ball will take 2 seconds to travel to the player's cannon position of the invader in the bottom row, and one extra second for each additional row.

vi) The player will be killed immediately when getting shot by the invader's cannon ball and the game terminates as the player is defeated. If all the invaders are terminated by the player, the game terminates as the player winning the game.

In this task, you are required to come up with an Open MPI design that can implement all of the above requirements.

Some questions to guide your design process:

1. How would you organise the MPI processes into a 2D Cartesian grid with Master/Slaves architecture?
2. How would you design the message sending & receiving mechanism to minimise delays while maximising speed-up?
3. How do you handle potential time synchronisation issues between different processes on different machines? How do you tackle potential game logic issues due to clock differences (e.g., one alien invader firing cannonballs faster than the other invaders)? Would creating a new MPI process performing clock synchronisation service be worthwhile for a real-time game?

Extra notes:

- Users should have an option to specify the grid dimension (e.g., 3x2 or 4x6) for the space invaders as command-line arguments, and you should not hardcode the grid.
- While this assessment does not require you to design a production-level gaming interface, it would be useful to have a function (called periodically) to print the state of all the alien invaders, the player's cannon, and all the cannon balls for debugging and presentation/demonstration.
- If you do not have sufficient cores/processors in your own machine to run the experiment, you are strongly recommended to compile and submit your program as a job on CAAS with the scheduler. In this case, you can store the player's input for simulation in a file and read the instructions in a file during simulation in the cluster.
- When executing ***mpirun***, you may need to use the ***oversubscribe*** option so that you can run more MPI processes than the number of cores you have in your own personal machine with limited cores. [Example] Specifying a grid dimension of 3x2 with 6 processes:

```
mpirun -np 6 -oversubscribe core_task 3 2
```

- When arguing why a particular design decision is chosen over the others, you should present convincing experimental evaluation/simulation results along with the argument.

Extended Task – Enhanced Game Logic

[Required for HD – 10% on demonstration and 10% on Q&A session]

In the extended task, you are required to change the original design and implement an enhancement of the game with the following new extra game logic/features:

a) Cannon ball logic: If alien invaders are getting shot by the player's cannon ball, the ball may have 20% chance directing to the left and killing the invader on the left (if exist & alive) immediately, 15% chance directing to the right and killing the invader on the right (if exist & alive) immediately, and 20% chance blocked by the shield of the invader. (i.e., only 45% of killing the target invader).

b) Invader reborn: For every dead alien invader, if the invader on the left and the invader on the right are both still alive, it will have a low chance (20% chance) to be reborn every 5 seconds.

c) Multi-players: At least 2 players will need to be supported. Each player will be represented as an individual MPI process with their own cannon. However, you will need to make sure the players' cannons do not overlap.

Similarly, some questions to guide your design process:

1. How would you redesign the message sending & receiving mechanism to cope with the changes correctly, and still be capable of minimize delays and maximising speed-up?
2. How would you handle the collision issue between two players' cannons in real time?
3. How do you handle potential time synchronisation issues between different processes on different machines? Again, how do you tackle potential game logic issues due to clock differences? Would your old design still work?