MONASH University

MESSAGE PASSING INTERFACE (MPI) III

OBJECTIVES

- Explore MPI Virtual Topologies and Master/Slave programs by splitting a communicator.

MARKS

- The lab is worthed 6 of the unit final mark.

INSTRUCTIONS

- Before class: Attempt all the tasks before class. Make sure you read the marking rubric and the lab pre-readings!
- During class:
    - You will be given 15 minutes to prepare your code for the demonstration period.
    - After you finish the preparation, you must submit all the files to Moodle.
    - During the demonstration period, the teaching staff will ask you questions on your code submission(s).
    - Late submission: 5% penalty (per day).
- Marks will not be awarded if you skip the class, do not make any submissions, or make an empty submission to Moodle (unless a special consideration extension has been approved).
- You are allowed to use search engines or AI tools to search for information and resources during pre-class preparation. However, search engines and AI tools are not allowed during the code presentation period.
- If you use any AI tools to prepare your answers, **you must declare during the interview and in your Moodle submission.**
- Always double check all your codes & answers.

ASSESSED TASKS

- Task 1 (10%) Task 2 (40%) Task 4 (50%)

LAB ACTIVITIES

Task 0 – 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!

Task 1 – Creating a 2D Cartesian grid with OpenMPI (10%)

	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”.

Submit your Open MPI solution for task 1 in “task1.c”.

```
#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;
}
```

Task 2 – IPC between adjacent processes in a Cartesian grid (40%)

With the help of your solution in Task 1, implement the following for Task 2.

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. [Extended task] How can we further optimise the basic program to *reduce overhead/improve networking performance*? For this extended task, you are required to implement an optimised version by further refining the basic version. You will need to also explain your reasons & rationale during the demonstration period, and write them down in less than 300 words.

Submit your Open MPI basic & optimised/extended solutions for task 2 in "task2_basic.c" and "task2_extended.c". Please upload your reasons & rationale for your optimised solution in "task2_report.pdf".

Task 3 - 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 few processes (often just one process) managing the tasks.

The manager is called the "master" and the others the "workers" or the "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
```

(You may assume the maximum length of all the messages in this task is 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).

Submit your Open MPI solution for task 3 in “task3.c”.

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;
}
```

Task 4 - 3D Cartesian Virtual Topology (50%)

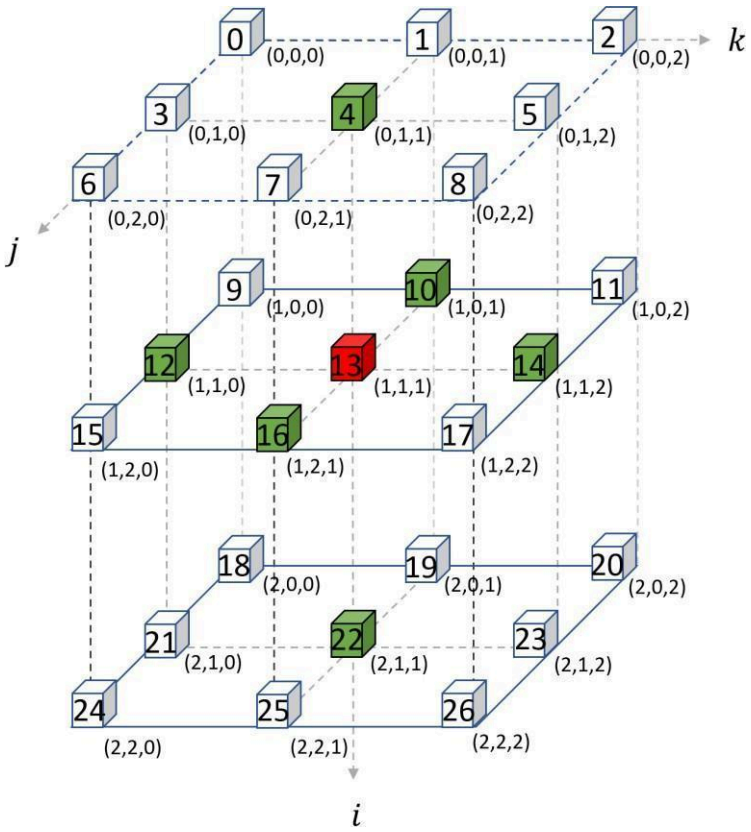

Figure 2: Sample 3D Cartesian (3×3×3). Mapping between a process rank and this Cartesian diagram is based on (i, j, k) indexing.
Image created by Ming Jie Lee.

In this task, you are required to use MPI virtual topology to implement a 3D Cartesian grid. Figure 2 illustrates an example. The figure also includes an example of a MPI process, rank 13 in red-coloured block, and its immediate adjacent (aka neighbourhood) processes labeled in green-coloured blocks.

Similar to Task 1, for this task each MPI process is required to print the following:

a) Current rank

b) Cartesian rank

c) Coordinates

d) List of immediate adjacent processes (left, right, top, bottom, front and rear)

You are also required to implement the following (similar to Task 2).

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 × 4 x 4 grid, we expect 64 unique log files. We assume if the log file is empty, 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 1 to 3 for at least 500 (or more) times. Make sure your program can generate at least a few records of matching prime numbers.

5. [Extended task] How can we further optimise the basic program to *reduce overhead/improve networking performance*? For this extended task, you are required to implement an optimised version by further refining the basic version. You will need to also explain your reasons & rationale during the demonstration period, and write them down in less than 300 words.

Submit your Open MPI basic & extended solutions for task 4 in "task4_basic.c" and "task4_extended.c". Please upload your reasons & rationale for your extended solution in "task4_report.pdf".

Note:

- Users must have an option to specify the grid dimension (e.g., 3x2x2 or 3x3x3) as a command line argument. You should not hardcode your program to tackle only one 3D grid.
- When executing ***mpirun***, you may need to use the ***oversubscribe*** option so that you could run more MPI processes than the number of cores you have in your own personal machine.
- [Example] Specifying a grid dimension of 3x2x2 with 12 processes:

  mpirun -np 12 -oversubscribe task4_basic 3 2 2

  If you do not have sufficient cores / processors in your own machine to run the experiment, please compile and submit your program as a job on CAAS.