MONASH University

FIT3143 Lab Week 10

Lecturers: ABM Russel (MU Australia) and Vishnu Monn (MU Malaysia)

MPI VIRTUAL TOPOLOGY + COMM SPLIT & PIPELINE COMPUTATION

OBJECTIVES

- The purpose of this lab is to first learn how to combine MPI Virtual Topologies with MPI Comm Split. Then, this lab explores integrating a thread into an MPI process.
- This lab also explores pipeline computation using MPI.

INSTRUCTIONS

- Download and set up the Linux VM [Refer to Lab Week 1]
- Setup eFolio (including Git) and share with tutor and partner [Refer to Lab Week 1]

TASK

DESCRIPTION:

- Combining MPI virtual topology functions and MPI Comm Split.
- Integrating a thread into an MPI process function.
- Design and implement an MPI pipeline computation program.

WHAT TO SUBMIT:

1. Algorithm or code description, analysis of results, screenshot of the running programs and git repository URL in the eFolio.
2. Code in the Git.

EVALUATION CRITERIA

- This Lab-work is part of grading
- Code compile without errors (2), well commented (2), lab-work questions fully answered (4), analysis/report is well formatted (2) = 10 marks

MONASH University

LAB ACTIVITIES (10 MARKS)

Task1 – Placing the Slaves into a virtual topology – Worked example

This task continues from Lab Week 09's Task 4. Here, the slaves are first placed in a 2D virtual topology (using MPI Cartesian functions). Each slave then sends a series of messages to the Master, which prints the message.

Sample solution:

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

#define MSG_EXIT 1
#define MSG_PRINT_ORDERED 2
#define MSG_PRINT_UNORDERED 3

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

int main(int argc, char **argv)
{
    int rank, size;
    MPI_Comm new_comm;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    MPI_Comm_split( MPI_COMM_WORLD,rank == size-1, 0, &new_comm);
// color will either be 0 or 1
    if (rank == size-1)
     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 world_comm, MPI_Comm comm)
{
     int        i, size, nslaves, firstmsg;
     char       buf[256], buf2[256];
```

```
     MPI_Status status;
     MPI_Comm_size(world_comm, &size );
     nslaves = size - 1;

     while (nslaves > 0) {
          MPI_Recv(buf, 256, MPI_CHAR, MPI_ANY_SOURCE,
MPI_ANY_TAG, world_comm, &status );
          switch (status.MPI_TAG) {
               case MSG_EXIT: nslaves--; break;
               case MSG_PRINT_UNORDERED:
                    fputs( buf, stdout );
               break;
               case MSG_PRINT_ORDERED:
                    firstmsg = status.MPI_SOURCE;
                    for (i=0; i<nslaves; i++) {
                         if (i == firstmsg)
                              fputs( buf, stdout );
                         else {
                              MPI_Recv( buf2, 256, MPI_CHAR, i,
MSG_PRINT_ORDERED, world_comm, &status );
                              fputs( buf2, stdout );
                         }
                    }
               break;
          }
     }
    return 0;
}

/* This is the slave */
int slave_io(MPI_Comm world_comm, MPI_Comm comm)
{
     int ndims=2, size, my_rank, reorder, my_cart_rank, ierr,
worldSize;
     MPI_Comm comm2D;
     int dims[ndims],coord[ndims];
     int wrap_around[ndims];
     char buf[256];

     MPI_Comm_size(world_comm, &worldSize); // size of the world
communicator
     MPI_Comm_size(comm, &size); // size of the slave communicator
     MPI_Comm_rank(comm, &my_rank);  // rank of the slave
communicator
     dims[0]=dims[1]=0;

     MPI_Dims_create(size, ndims, dims);
     if(my_rank==0)
          printf("Slave Rank: %d. Comm Size: %d: Grid Dimension =
[%d x %d] \n",my_rank,size,dims[0],dims[1]);

     /* create cartesian mapping */
```

```
    wrap_around[0] = 0;
    wrap_around[1] = 0; /* periodic shift is .false. */
    reorder = 0;
    ierr =0;
    ierr = MPI_Cart_create(comm, ndims, dims, wrap_around,
reorder, &comm2D);
    if(ierr != 0) printf("ERROR[%d] creating CART\n",ierr);

    /* find my coordinates in the cartesian communicator group */
    MPI_Cart_coords(comm2D, my_rank, ndims, coord); //
coordinated is returned into the coord array
    /* use my cartesian coordinates to find my rank in cartesian
group*/
    MPI_Cart_rank(comm2D, coord, &my_cart_rank);

/*
    printf("Global rank (within slave comm): %d. Cart rank: %d.
Coord: (%d, %d).\n", my_rank, my_cart_rank, coord[0], coord[1]);
    fflush(stdout);
*/

    sprintf( buf, "Hello from slave %d at Coordinate: (%d,
%d)\n", my_rank, coord[0], coord[1]);
    MPI_Send( buf, strlen(buf) + 1, MPI_CHAR, worldSize-1,
MSG_PRINT_ORDERED, world_comm );

    sprintf( buf, "Goodbye from slave %d at Coordinate: (%d,
%d)\n", my_rank, coord[0], coord[1]);
    MPI_Send(buf, strlen(buf) + 1, MPI_CHAR, worldSize-1,
MSG_PRINT_ORDERED, world_comm);

    sprintf(buf, "Slave %d at Coordinate: (%d, %d) is exiting\n",
my_rank, coord[0], coord[1]);
    MPI_Send(buf, strlen(buf) + 1, MPI_CHAR, worldSize-1,
MSG_PRINT_ORDERED, world_comm);
    MPI_Send(buf, 0, MPI_CHAR, worldSize-1, MSG_EXIT,
world_comm);

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

MONASH University

Task2 – Adding a thread as an asynchronous activity at the Master – Worked Example

This task is based on modifying the sample solution code from Task 1 above. Move the **bolded region of the `while` loop code** in the sample solution of Task 1 into a thread function. This means that the `master_io()` function does the following:

a) Creates a thread
b) Waits for the thread to complete
c) Exits

The thread function implements the bolded while loop code (i.e., waiting for messages from the slaves and printing these messages). Make sure to pass the necessary values from the `master_io()` function to the thread.

You can use the POSIX thread library to create the thread here. You could also consider using OpenMP as an asynchronous thread.

Note: You may feel that this task is unnecessary as the Task 1 program code works just fine. However, the aim of Task 2 here is to provide you some basic exposure on using thread with MPI.

Sample solution:

```
// Sample solution focuses on the thread function and master_io
function.

void* ProcessFunc(void *pArg) // Common function prototype
{
     int i = 0, size, nslaves, firstmsg;
     char buf[256], buf2[256];
     MPI_Status status;
     MPI_Comm_size(MPI_COMM_WORLD, &size );

     int* p = (int*)pArg;
     nslaves = *p;

     while (nslaves > 0) {
          MPI_Recv(buf, 256, MPI_CHAR, MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD, &status );
          switch (status.MPI_TAG) {
               case MSG_EXIT: nslaves--; break;
               case MSG_PRINT_UNORDERED:
                    printf("Thread prints: %s", buf);
                    fflush(stdout);
               break;
               case MSG_PRINT_ORDERED:
                    firstmsg = status.MPI_SOURCE;
                    for (i=0; i<size-1; i++) {
                         if (i == firstmsg){
                              printf("Thread prints: %s", buf);
                              fflush(stdout);
                         }else {
```

```
                                MPI_Recv( buf2, 256, MPI_CHAR, i,
MSG_PRINT_ORDERED, MPI_COMM_WORLD, &status );
                                printf("Thread prints: %s", buf2);
                                fflush(stdout);
                            }
                        }
                    break;
                }
            }


        return 0;
}

int master_io(MPI_Comm world_comm, MPI_Comm comm)
{
        int size, nslaves;
        MPI_Comm_size(world_comm, &size );
        nslaves = size - 1;

        pthread_t tid;
        pthread_create(&tid, 0, ProcessFunc, &nslaves); // Create the
thread
        pthread_join(tid, NULL); // Wait for the thread to complete.

    return 0;
}
```

Task3 – Simple pipeline computing using MPI – Worked example

A series of raw experimental results have been stored into the ***ExpResults.txt*** file, as shown in Figure T3. The first element in ***ExpResults.txt*** represents the number of experiment results in this file. Subsequent elements in this file represent the respective experiment results.

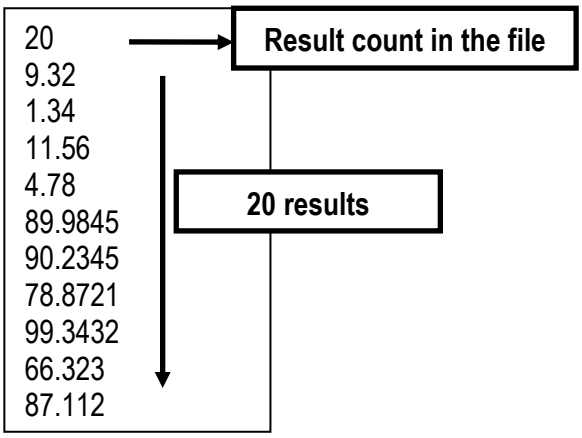

Figure T3: Content of *ExpResult.txt*

Each piece of result from this file, x_0, has be subjected to further calculations as per the following mathematical expressions:

$$x_1 = x_0 - 4x_0 + 7$$

$$x_2 = x_1^3 + \sin(\frac{x_1}{8})$$

$$x_3 = 2x_2^4 + \cos(4x_2) + 3\pi$$

$$x_4 = 3x_3^2 - 2x_3 + \frac{\tan(x_3)}{3}$$

Using the C programming language with a parallel programming implementation using the Message Passing Interface (MPI):

Write a program to apply the content of ***ExpResults.txt*** into the series of equations as aforementioned such that the result for each data content is represented by x_4.

Only the root rank is permitted to access the data content of ***ExpResults.txt*** and read each datum one by one. Each datum read from this file is stored into variable x_0. The root node then calculates x_1 based on the value of x_0. The result of x_1 is then transmitted to the subsequent node, which calculates x_2. This process continues in a **parallel pipeline structure** until the last node calculates x_4.

In addition, only the root node will print out the results of x_4 for each datum read from file ***ExpResults.txt***. To achieve this design, the last node will store a list of calculated x_4 results before sending this list back to the root node for result printout. Use dynamic memory allocation to create a list based on the number of elements in the file.

Note: The total number of nodes in the pipeline architecture should be four, with x_1, x_2, x_3 & x_4 representing the first, second, third and fourth nodes respectively. Please create your own ExpResults.txt file based on the format as seen in Figure T3. You need not create a large number of entries for your version of ExpResults.txt. A smaller number of entries would suffice to test the program.

Sample solution:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <memory.h>
#include "mpi.h"

#define SENTINEL 0.0

int main(int argc, char *argv[])
{
     FILE *pInfile;
     double x0, x1, x2, x3, x4;
     double *pX4Buff = NULL;
     float x;
     int fileElementCount = 0;
     int counter = 0;

     int my_rank;
```

```
     int p;
     MPI_Status status;

     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
     MPI_Comm_size(MPI_COMM_WORLD, &p);

     switch (my_rank)
     {
          case 0:
          {
               pInfile = fopen("ExpResults.txt","r");
               fscanf(pInfile, "%d", &fileElementCount);

               pX4Buff = (double*)malloc(fileElementCount *
sizeof(double));
               memset(pX4Buff, 0, fileElementCount *
sizeof(double));

               // Send the counter to the last process
               MPI_Send(&fileElementCount, 1, MPI_INT, (p - 1), 0,
MPI_COMM_WORLD);

               // Read each element from the file
               while(counter < fileElementCount)
               {
                    fscanf(pInfile, "%f", &x);
                    x0 = x;
                    x1 = x0 - (4 * x0) + 7;
                    MPI_Send(&x1, 1, MPI_DOUBLE, 1, 0,
MPI_COMM_WORLD);
                    counter++;
               }

               // File end, send a SENTINEL value to complete
calculation
               fclose(pInfile);
               pInfile = NULL;

               x1 = SENTINEL;
               MPI_Send(&x1, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);

               // Wait for buffer from last node
               MPI_Recv((void*)pX4Buff, counter, MPI_DOUBLE, (p -
1), 0, MPI_COMM_WORLD, &status);

               // Print results
               for(int i = 0; i < counter; i++)
               {
                    printf("Result[%d]: %g\n", i, pX4Buff[i]);
               }
               free(pX4Buff);
```

```
                pX4Buff = NULL;
                break;
           }
           case 1:
           {
                do
                {
                     MPI_Recv(&x1, 1, MPI_DOUBLE, 0, 0,
MPI_COMM_WORLD, &status);
                     if(x1 != SENTINEL)
                     {
                          x2 = pow(x1, 3) + sin(x1/8);
                     }
                     else
                     {
                          x2 = SENTINEL;
                     }
                     MPI_Send(&x2, 1, MPI_DOUBLE, 2, 0,
MPI_COMM_WORLD);
                } while (x1 != SENTINEL);
                break;
           }
           case 2:
           {
                do
                {
                     MPI_Recv(&x2, 1, MPI_DOUBLE, 1, 0,
MPI_COMM_WORLD, &status);
                     if(x2 != SENTINEL)
                     {
                          x3 = (2 * pow(x2, 4)) + cos(4 * x2) + (3
* M_PI);
                     }
                     else
                     {
                          x3 = SENTINEL;
                     }
                     MPI_Send(&x3, 1, MPI_DOUBLE, 3, 0,
MPI_COMM_WORLD);
                } while (x2 != SENTINEL);
                break;
           }

           case 3:
           {
                // Get the file element count first
                MPI_Recv(&fileElementCount, 1, MPI_DOUBLE, 0, 0,
MPI_COMM_WORLD, &status);
                pX4Buff = (double*)malloc(fileElementCount *
sizeof(double));
                memset(pX4Buff, 0, fileElementCount *
sizeof(double));
```

```
                // Now, receive the pipelined data
                counter = 0;
                do
                {
                     MPI_Recv(&x3, 1, MPI_DOUBLE, 2, 0,
MPI_COMM_WORLD, &status);
                     if(x3 != SENTINEL)
                     {
                          x4 = (3 * pow(x3, 2)) - (2 * x3) +
(tan(x3) / 3);

                          // Save the result into buffer &
increment the buffer counter
                          pX4Buff[counter] = x4;
                          counter++;
                     }
                } while (x3 != SENTINEL);

                // End of file reached, send the buffer back to the
root
                if(counter > 0)
                {
                     MPI_Send(pX4Buff, counter, MPI_DOUBLE, 0, 0,
MPI_COMM_WORLD);
                }
                // Clean up
                free(pX4Buff);
                pX4Buff = NULL;
                break;
           }
           default:
           {
                printf("Process %d unused.\n",my_rank);
                break;
           }
     }

     MPI_Finalize();

  return 0;
}
```

MONASH University

Task 4 – Pipeline computing exercise using MPI

In mathematics, a quadratic equation represents a univariate polynomial equation of the second degree. A general quadratic equation can be described as:

$$ax^2 + bx + c = 0 \qquad (1.1)$$

where *x* represents the unknown variable and *a*, *b* and *c* are the quadratic coefficients ($a \neq 0$). A quadratic equation with real and complex coefficients has two solutions, called roots (x_1 and x_2).

The discriminant, *d*, is computed as: $d = b^2 - 4ac$. If *d* is positive, the quadratic equation has two distinct real roots (i.e., $x_1 \neq x_2$) such that:

$$x_1 = \frac{-b+\sqrt{d}}{2a}, x_2 = \frac{-b-\sqrt{d}}{2a} \qquad (1.2)$$

If *d* is zero, the quadratic equation has only one real root (i.e., $x_1 = x_2$) such that:

$$x_1 = x_2 = \frac{-b}{2a} \qquad (1.3)$$

If *d* is negative, the quadratic equation has two distinct complex roots (i.e., $x_1 \neq x_2$) such that:

$$x_1 = \frac{-b}{2a} + \frac{i\sqrt{|d|}}{2a}, x_2 = \frac{-b}{2a} - \frac{i\sqrt{|d|}}{2a} \qquad (1.4)$$

Figure T4-1 illustrates the content of a text file, *quad.txt*, which contains a set of quadratic coefficients. The first row element in *quad.txt* represents the number of coefficients (*a*, *b* and *c*) rows in this file. The second row displays the legend text for these coefficients and the third row onwards contains the coefficient content.

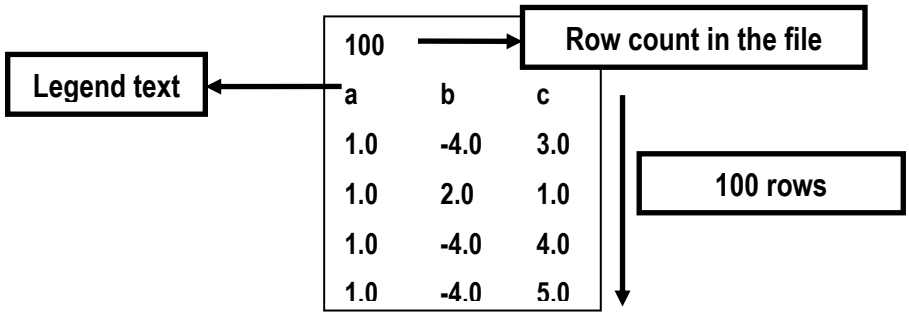

Figure T4-1: Content of *quad.txt*

Using the C programming language with a parallel programming implementation using the Message Passing Interface (MPI):

a) Write a program to compute the quadratic roots of each row coefficient in *quad.txt* file.

i. Only the root rank (or first node) is permitted to access the *quad.txt* file.
ii. The root rank reads coefficients of each row from *quad.txt* file and computes the discriminant, *d*, which is then transmitted to the subsequent node along with the *a* and *b* coefficients.

b) Continuing from part (a), the second node receives the computed *d* and *a* and *b* coefficients (per row) from the first node and computes the root values.

i. This node computes x_1 and x_2 based on the computed value of *d*.
ii. Note: If $d < 0$, the roots are calculated as x_{1_real}, x_{1_img} and x_{2_real}, x_{2_img}.
iii. The computed root values are then transmitted to the third node.

c) Continuing from part (b), the third node receives the computed root values from the second node and writes these root values into a new text file, *roots.txt*. Figure T4-2 illustrates a sample content of the computed root value in the *roots.txt* file.

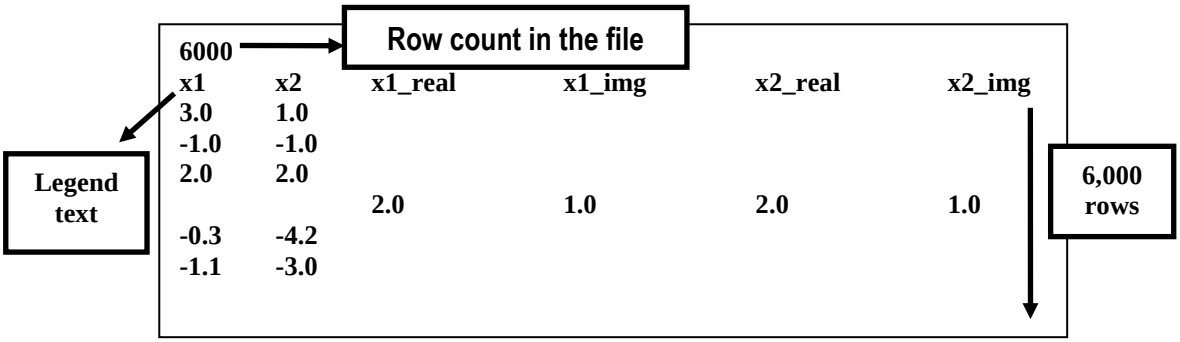

Figure T4-2: Content of *roots.txt*

Complete parts (a), (b) and (c). Use a **parallel pipeline structure** in reading the coefficients from the file, computing the roots and writing the computed roots into file.

Note: *The total number of nodes in this pipeline architecture should be three (The first node reads a row coefficient and computes d, the second node computes the roots and the third node writes the computed roots into a new file). Create your own version of quad.txt as seen in Figure T4-1. You need not create a large number of entries for your version of quad.txt. A smaller number of entries would suffice to test the program.*

You may refer to the following C code to get you started.

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

int main()
{
      FILE *pInfile;
      float a_coeff, b_coeff, c_coeff, x1, x2, disc;
      float x1r, x1i, x2r, x2i;
      int fileElementCount = 0, constNeg = -1;;

      int my_rank;
      int p;
      MPI_Status status;

      MPI_Init(&argc, &argv);
```

```
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
        MPI_Comm_size(MPI_COMM_WORLD, &p);

        // WRITE PART(a) HERE

        switch (my_rank){
              case 0:{
                    // CONTINUE WITH PART (a) HERE
                    break;
              }
              case 1:{
                    // WRITE PART (b) HERE
                    break;
              }
              case 2:{
                    // WRITE PART (c) HERE
                    break;
              }
        }
        MPI_Finalize();
        return 0;
}
```

OPTIONAL BONUS ACTIVITIES (2 MARKS)

Develop a pipeline solution to compute *sin* according to

$$\sin\theta = \theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} - \frac{\theta^7}{7!} + \frac{\theta^9}{9!} - \ldots$$

A series of values represent the input, θ_0, θ_1, θ_2 ,θ_3 .....