MONASH University

FIT3143 Lab Week 6

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

FURTHER MESSAGE PASSING INTERFACE

OBJECTIVES

- The purpose of this lab is to introduce you to 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:

- Become familiar with MPI
- Practice parallel algorithm design and development
- Practice buffer transfer between MPI processes

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

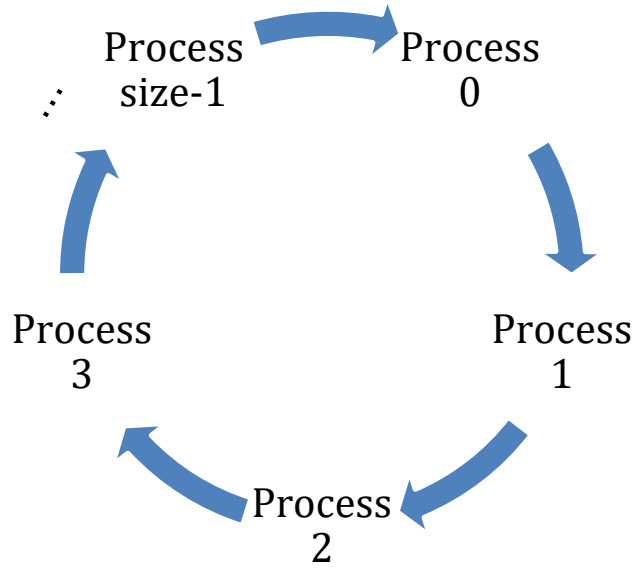
- This Lab-work is part of grading
- Code compile without errors (2), well commented (2), lab-work questions fully answered (4), analysis/report is well formatted (2) = 10 marks
- 2 bonus marks for challenging activities

MONASH University

LAB ACTIVITIES (10 MARKS)

1. Sending in a ring (broadcast by ring)

Write an MPI program that takes data from process zero and sends it to all other processes by sending it in a ring. That is, process *i* should receive the data and send it to process *i*+1, until the last process is reached. The last process then sends the data back to process zero. Each MPI process prints out the received data. Use a loop to repeat the cycle until a sentinel value is specified to exit the program.



The following starter code is provided. This code is incomplete. Complete the code. Compile and execute the code using at least four MPI processes. Observe and display your results.

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

int main(int argc, char **argv)
{
     int rank, s_value, r_value, size;
     MPI_Status status;
     MPI_Init( &argc, &argv );
     MPI_Comm_rank( MPI_COMM_WORLD, &rank );
     MPI_Comm_size( MPI_COMM_WORLD, &size );
     do {
          if (rank == 0) {
               printf("Enter a round number: ");
               fflush(stdout);
               scanf( "%d", &s_value );
               // Add your code here
               printf( "Process %d got %d from Process %d\n",
rank, r_value, size - 1);
               fflush(stdout);
          }
          else {
```

```
                // Add your code here

                printf( "Process %d got %d from Process %d\n",
rank, r_value, rank - 1);
                fflush(stdout);
            }
        } while (r_value >= 0);

        MPI_Finalize( );
        return 0;
}
```

2. Prime Search using Message Passing Interface

Let's revisit the prime search problem which you previously worked on.

In Week 03, you were required to:

a) Write a serial C program to search for prime numbers which are less than an integer, *n*, which is provided by the user. The expected program output is a list of all prime numbers found, which is written into a single text file (e.g., *primes.txt*).

b) Measure the time required to search for prime numbers less than an integer, *n* when *n = 10,000,000* (i.e., t_s). Calculate the theoretical speed up using Amdahl's law when *p = 4*, with *p = number of processes (or threads)*.

c) Write a parallel version of your serial code in C utilizing POSIX Threads. Here, design and implement a parallel partitioning scheme which distributed the workload among the threads. Compare the performance of the serial program (t_s) in part (a) with that of the parallel program (t_p) in part (b) for a fixed number of processors or threads (e.g., *p = 4*). Calculate the actual speed up, *S(p)*.

In this week's lab activity, we shall continue from where we left off in Week 03. In detail:

d) Implement a parallel version of your serial code in C using Message Passing Interface (MPI).

   i) The root process will prompt the user for the *n* value (*n = 10,000,000* (i.e., t_s).). The specified *n* value will then be disseminated to other MPI processes to calculate the prime numbers. Each process (including root process) computes the prime number (based on the equal or varied workload distribution per node) and writes the computed prime number into text files.

      The name of the text files should include the node rank value (e.g. *process_0.txt, process_1.txt, process_2.txt,* etc.). Execute the compiled program using at least four MPI processes (i.e., *p = 4*) using your virtual machine or physical computer.

   ii) Measure the overall time required to search for prime numbers less than an integer, *n* when *n = 10,000,000.* Compare your results with the serial version in part (a) and calculate the actual speed up (i.e., $s_{actual}(p) = \frac{t_s}{t_p}$). Analyse and compare the actual speed up with the theoretical speed up for an increasing

number of MPI processes and/or *n*. You can either tabulate your results or plot a chart when analysing the performance of the serial and parallel programs.

iii) In addition, write your observation comparing the actual speed up between part (d) and part (c) whether the speed up using MPI is any better than using POSIX threads.

e) Modify part (d) such that each MPI process returns the computed prime numbers to the root process.

i) The root process receives the computed prime numbers and prints this number into a single text file. Note that only the root process writes the computed prime numbers to a text file. All other MPI processes (inclusive of root process) compute the prime numbers based on the assigned workload distribution.

ii) Repeat parts (d)(ii) and (d)(iii) steps for Part (e) and write your observation.

Note: You may opt to use different algorithms to search for prime numbers. However, please ensure that you implement both the serial and parallel versions of the algorithm in C based on the aforementioned specifications.

OPTIONAL BONUS ACTIVITIES (2 MARKS)

3. Executing the prime search algorithm on MonARCH Cluster

a) Please refer to the instructions in Week 05 of Moodle for FIT3143 on how to access MonARCH. Ensure that you have tried out the sample example code in MonARCH.

b) Based on your serial and parallel code implementations of the prime search algorithm:

i) Compile and execute the serial code in MonARCH. Measure the time taken to complete the task (i.e., *ts*).

ii) Compile and execute the parallel code using POSIX threads in MonARCH. Measure the time taken to complete the task (i.e., *tp*) for 16 threads on a single compute node.

iii) Compile and execute the parallel code using MPI in MonARCH. Measure the time taken to complete the task (i.e., *tp*) for 16 MPI processes on a single compute node (i.e., single server).

iv) Repeat part (iii) with 32 MPI processes on two compute nodes (i.e., a server cluster with 16 MPI processes on a single server).

Include screenshots of step i) to step iv) executions on MonARCH Cluster and tabulate all actual speed ups.

Hints: You need to revise the code to no longer prompt the user for the value of *n* at runtime. Instead, the value of *n* is now passed as a command line argument into the application.