MONASH University

FIT3143 Lab Week 5

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

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 Simple Parallel Data Structure
- Practice parallel algorithm design and development

WHAT TO SUBMIT:

1. Screenshot of the running programs and git repository URL in the eFolio. 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

(BASED ON TUTORIAL BY WILLIAM GROPP AND EWING LUSK)

1. Getting Started with “Hello World” – A Worked Example

Write a program that uses MPI and has each MPI process print

```
Hello world from process i of n
```

using the rank in MPI_COMM_WORLD for i and the size of MPI_COMM_WORLD for n. You can assume that all processes support output for this example.

Do take note the order that the output appears in. Depending on your MPI implementation, characters from different lines may be intermixed.

You may want to use these MPI routines in your solution:

```
MPI_Init MPI_Comm_size MPI_Comm_rank MPI_Finalize
```

Note: Question 1 is designed to ensure that you are able to compile and execute a simple C code with MPI. Display a screenshot of the executed programme in the eFolio sheet.

Sample Solution

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

int main( argc, argv )
int  argc;
char **argv;
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    printf( "Hello world from process %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

Makefile

```
# Generated automatically from Makefile.in by configure.
ALL: helloworld

helloworld: helloworld.c
     mpicc -o helloworld helloworld.c

clean:
     /bin/rm -f helloworld *.o
```

Output

```
% mpicc -o helloworld helloworld.c
% mpirun -np 4 helloworld
Hello world from process 0 of 4
Hello world from process 3 of 4
Hello world from process 1 of 4
Hello world from process 2 of 4
%
```

2. Shared Data – A Worked Example

A common need is for one process to get data from the user, either by reading from the terminal or command line arguments, and then to distribute this information to all other processors.

Write a program that reads an integer value from the terminal and distributes the value to all of the MPI processes. Each process should print out its rank and the value it received. Values should be read until a negative integer is given as input.

You may find it helpful to include a `fflush(stdout);` after the `printf` calls in your program. Without this, output may not appear when you expect it.

You may want to use these MPI routines in your solution:

`MPI_Init MPI_Comm_rank MPI_Bcast MPI_Finalize`

Sample Solution

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

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

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

     int val = -1;
     do{
          if(my_rank == 0)
          {
               printf("Enter a round number (> 0 ): ");
               fflush(stdout);
               scanf("%d", &val);
          }
          MPI_Bcast(&val, 1, MPI_INT, 0, MPI_COMM_WORLD); // This
function has to visible to all processes.
          printf("Processors: %d. Received Value: %d\n", my_rank,
val);
          fflush(stdout);
     }while(val > 0);

     MPI_Finalize();
     return 0;
}
```

3. Using MPI datatypes to share data – A Worked Example

In this task, you will modify your argument broadcast routine to communicate different data types with a single MPI broadcast (`MPI_Bcast`) call. Have your program read an integer and a double-precision value from standard input (from process 0, as before), and communicate this to all of the other processes with an `MPI_Bcast` call. Use MPI datatypes.

Have all processes exit when a negative integer is read.

If you are not familiar with structs in C programming, have a quick look here.

You may want to use these MPI routines in your solution:

`MPI_Get_address MPI_Type_create_struct MPI_Type_commit MPI_Type_free MPI_Bcast`

Sample Solution

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

struct valuestruct {
     int a;
     double b;
} ;

int main(int argc, char** argv)
{
     struct valuestruct values;
     int myrank;
     MPI_Datatype Valuetype;
     MPI_Datatype type[2] = { MPI_INT, MPI_DOUBLE };
     int blocklen[2] = { 1, 1};
     MPI_Aint disp[2];

     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

     MPI_Get_address(&values.a, &disp[0]);
     MPI_Get_address(&values.b, &disp[1]);

     //Make relative
     disp[1]=disp[1]-disp[0];
     disp[0]=0;
```

```
      // Create MPI struct
      MPI_Type_create_struct(2, blocklen, disp, type, &Valuetype);
      MPI_Type_commit(&Valuetype);

      do{
          if (myrank == 0){
           printf("Enter an round number (>0) & a real number: ");
           fflush(stdout);
           scanf("%d%lf", &values.a, &values.b);
          }
          MPI_Bcast(&values, 2, Valuetype, 0, MPI_COMM_WORLD);
          printf("Rank: %d. values.a = %d. values.b = %lf\n",
myrank, values.a, values.b);
          fflush(stdout);
      }while(values.a > 0);

      /* Clean up the type */
      MPI_Type_free(&Valuetype);
      MPI_Finalize();
      return 0;
}
```

4. Using MPI_Pack to share data

In this task, you will modify your argument broadcast routine to communicate different data types by using `MPI_Pack` and `MPI_Unpack`.

Have your program read an integer and a double-precision value from standard input (from process 0, as before), and communicate this to all of the other processes with an `MPI_Bcast` call. Use `MPI_Pack` to pack the data into a buffer (for simplicity, you can use `char packbuf[100];` but consider how to use `MPI_Pack_size` instead).

Note that `MPI_Bcast`, unlike `MPI_Send/MPI_Recv` operations, requires that exactly the same amount of data be sent and received. Thus, you will need to make sure that all processes have the same value for the count argument to `MPI_Bcast`.

Have all processes exit when a negative integer is read.

You may want to use these MPI routines in your solution:

`MPI_Pack MPI_Unpack MPI_Bcast`

5. Using MPI_Reduce to approximate the value of Pi

π (sometimes written pi) is a mathematical constant whose value is the ratio of any circle's circumference to its diameter in Euclidean space; this is the same value as the ratio of a circle's area to the square of its radius. It is approximately equal to 3.141593 in the usual decimal notation (see the table for its representation in some other bases).

This exercise presents a simple program to determine the value of pi, based on the following algorithm:

Algorithm: Find the integration of pi between 0 & 1. This integral is approximated by a sum of *N* intervals. The approximation to the integral in each interval is: $\frac{1}{N}\times\frac{4}{(1+x*x)}$. In computational terms, this algorithm can be further represented as:

$$\boxed{\frac{1}{N}\times\left(\sum_{i=0}^{N-1}\frac{4}{1+\left(\frac{2i+1}{2N}\right)^2}\right)}$$

A serial based C code implementation of this algorithm is provided as follows:

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

static long N = 100000000;

int main(int argc, char* argv[])
{
     int i;
     double sum = 0.0;
     double piVal;
     struct timespec start, end;
     double time_taken;

     // Get current clock time.
     clock_gettime(CLOCK_MONOTONIC, &start);

     for(i = 0; i < N; i++)
     {
          sum += 4.0 / (1 + pow((2.0 * i + 1.0)/(2.0 * N), 2));
     }
     piVal = sum / (double)N;

     // Get the clock current time again
```

```
    // Subtract end from start to get the CPU time used.
    clock_gettime(CLOCK_MONOTONIC, &end);
    time_taken = (end.tv_sec - start.tv_sec) * 1e9;
    time_taken = (time_taken + (end.tv_nsec - start.tv_nsec)) *
1e-9;

    printf("Calculated Pi value (Serial-AlgoI) = %12.9f\n",
piVal);
    printf("Overall time (s): %lf\n", time_taken);    // ts

    return 0;
}
```

Based on the algorithm and serial C code in the preceding page:

a) Compile and run the serial code. Measure the overall time taken to complete the computation (i.e., *ts*).

b) Implement a parallel version of this algorithm using Message Passing Interface (MPI). The root rank will prompt the user for the *N* value. The specified *N* value will then be disseminated to other processors to calculate the value of Pi based. Apply a data parallel design based on the value of *N* and the number of MPI processes. Measure the overall time taken (i.e., *tp*) for large values of *N* (e.g., *N* = 100,000,000) and compute the actual speed up (if any).

   You may want to use these MPI routines in your solution:

   `MPI_Bcast` and **`MPI_Reduce`**.

Note: You may opt to use different algorithms to approximate the value of Pi. However, please ensure that you implement parallel version (using MPI) of the algorithm in C.