MONASH University

MESSAGE PASSING INTERFACE (MPI)

OBJECTIVES

- The purpose of this lab is to introduce you to Message Passing Interface (MPI)

MARKS

- The lab is worthed 5 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 20 minutes to prepare your code for the demonstration period.
    - When you have finished preparing the code, you are required to submit them to Moodle.
    - During the demonstration period, the teaching staff will ask you questions on your code submission(s).
- 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.
- Always double test all your codes & written answers. Make sure the codes can be run on your laptop.

ASSESSED TASKS

- Task 2 (10%) [task2.c]
- Task 3 (20%) [task3.c]
- Task 4 (20%) [task4.c]
- Task 5 (50%) [task5.c & answer.pdf]

LAB ACTIVITIES

(BASED ON TUTORIAL BY WILLIAM GROPP AND EWING LUSK)

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 – Getting Started with "Hello World" – A Worked Example (Not assessed)

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 of 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: Task 1 is really just designed to ensure that you are able to compile and execute a simple C code with MPI and test your Linux environment. Please also spend time looking up these methods in the manual page (Task 0) and study their functionalities again!

Please copy/type in a solution below. For the main C file:

```
#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;
}
```

For the Makefile:

```
ALL: helloworld

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

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

If you don't like to use Makefile, here is the compilation command:

```
mpicc -o helloworld helloworld.c
```

Command to run after compilation (assuming 4 processes):

```
mpirun -np 4 helloworld
```

Example output (note: The line ordering can be different):

```
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
```

Task 2 – Shared Data – A Worked Example (10%)

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`

Please copy/type in the code template below. There is one line of code missing, please fill in the missing code and submit as "task2.c"!

```
#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);
          }
          // Please insert one line of code here
          printf("Processors: %d. Received Value: %d\n", my_rank,
val);
          fflush(stdout);
     }while(val > 0);
```

```
    MPI_Finalize();
  return 0;
}
```

Task 3 – Using MPI datatypes to share data – A Worked Example (20%)

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`

Please copy/type in the code template below.

There are two lines of code missing, please fill in the missing code and submit as "task3.c"!

```
#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
      // Please insert one line of code here (Hint:
 MPI_Type_create_struct)
      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);
          }
      // Please insert one line of code here (Hint: MPI_Bcast)
          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;
 }
```

Task 4 – Using MPI_Pack to share data (20%)

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`

Please copy/type in the code template below.

There are 4 lines of code missing, please fill in the missing code and submit as "task4.c"!

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

int main(){

   int my_rank;
   struct timespec ts = {0, 50000000L}; /* wait 0 sec and 5^8 nanosec */

   int a; double b;
   char *buffer; int buf_size, buf_size_int, buf_size_double, position =
0;

   MPI_Init(NULL, NULL);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   // Determine buffer size
   MPI_Pack_size(1, MPI_INT, MPI_COMM_WORLD, &buf_size_int);
   MPI_Pack_size(1, MPI_DOUBLE, MPI_COMM_WORLD, &buf_size_double);

   buf_size = buf_size_int + buf_size_double; // Increase the buffer size

   // Allocate memory to the buffer
   buffer = (char *) malloc((unsigned) buf_size);

   do{
        if(my_rank == 0){
             nanosleep (&ts, NULL);

             printf("Enter an round number (>0) & a real number: ");
             fflush(stdout);
             scanf("%d %lf", &a, &b);

             position = 0; // Reset the position in buffer

             // Pack the integer a into the buffer
             // Please insert one line of code here (Hint: a
          MPI_Pack function call)

             // Pack the double b into the buffer
             // Please insert one line of code here (Hint:
          another MPI_Pack function call)

        }

        // Broadcast the buffer to all processes
        MPI_Bcast(buffer, buf_size, MPI_PACKED, 0, MPI_COMM_WORLD);


        position = 0; // Reset the position in buffer in each iteration

        // Unpack the integer a from the buffer
        // Please insert one line of code here (Hint: a
       MPI_Unpack function call)
```

```
        // Unpack the double b from the buffer
        // Please insert one line of code here (Hint: another
        MPI_Unpack function call)


        printf("[Process %d] Received values: values.a = %d, values.b =
%lf\n", my_rank, a, b);
        fflush(stdout);

        MPI_Barrier(MPI_COMM_WORLD);
   }while(a > 0);

   /* Clean up */
   free(buffer);
   MPI_Finalize();

   return 0;
}
```

Task 5 – Using MPI_Reduce to approximate the value of Pi (50%)

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:

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

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) Make sure you can 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) **and submit as "task5.c"**.
   - The root rank will prompt the user for the *N* value.
   - The specified *N* value, together with other necessary information, will then be disseminated to other processors to calculate part of the value of Pi.
   - Apply a data parallel design based on the value of *N* and the number of MPI processes.
   - You will need to 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).

   Hint: These two MPI functions are extremely useful: `MPI_Bcast` and `MPI_Reduce`.

c) Compare the performance of the serial and parallel implementations of your Pi approximation code. Write down the **speedup** and give **at least two reasons** on why the actual speed up is different from the theoretical speed up. Submit your answers in a PDF file "answer.pdf".

Note: There are many algorithms/approaches to approximate the integral. Some may be easy to parallelize while some can be challenging. If one algorithm doesn't work, please try another approach! Make sure you implement both the serial and parallel version in C/C++ (with Open MPI).