MONASH University

FIT3143 Lab Week 7

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

PARALLEL DATA STRUCTURES & NON-BLOCKING COMMUNICATION USING MPI

OBJECTIVES

- Design and develop parallel algorithms for various parallel computing architectures
- Analyse and evaluate the performance of parallel algorithms

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:

- Practice parallel algorithm design and development
- Practice non-blocking communication between MPI processes
- Analyse the performance of blocking and non-blocking MPI communication

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. Performance analysis (if any).
2. Code in the Git.

EVALUATION CRITERIA:

- **This Lab-work is not assessed**. Nevertheless, we do encourage you to attempt the questions in this lab.

LAB ACTIVITIES (10 MARKS)

1. A Parallel Data Structure (Worked Example)

This task implements a simple parallel data structure. This structure is a two-dimension regular mesh of points, divided into slabs, with each slab allocated to a different MPI process. In the simplest C form, the full data structure is

```
double x[maxn][maxn];
```

and we want to arrange it so that each process has a local piece:

```
double xlocal[maxn/size][maxn];
```

where size is the size of the communicator (e.g., the number of MPI processes).

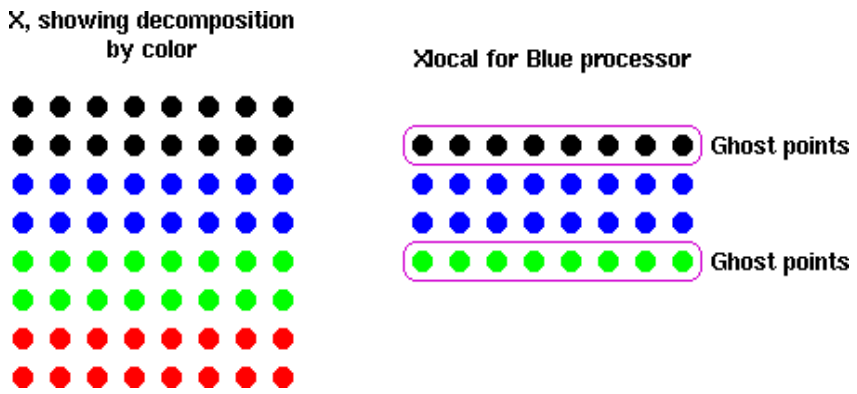
If that was all that there was to it, there wouldn't be anything to do. However, for the computation that we're going to perform on this data structure, we'll need the adjacent values. That is, to compute a new `x[i][j]`, we will need


```
x[i][j+1] x[i][j-1] x[i+1][j] x[i-1][j]
```

The last two of these could be a problem if they are not in `xlocal` but are instead on the adjacent processes. To handle this difficulty, we define ghost points that we will contain the values of these adjacent points.

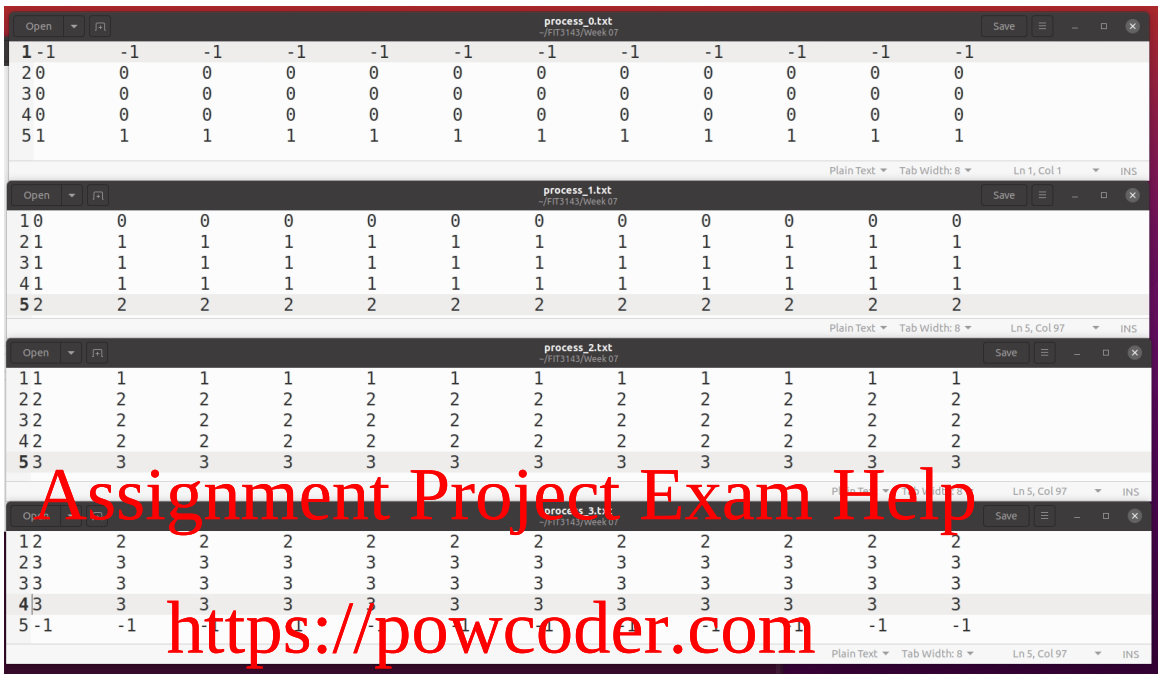
Write code to copy divide the array `x` into equal-sized strips and to copy the adjacent edges to the neighboring processes. Assume that `x` is `maxn` by `maxn`, and that `maxn` is evenly divided by the number of processes. For simplicity, you may assume a fixed size array and a fixed (or minimum) number of processors.

To test the routine, have each process fill its section with the rank of the process, and the ghost points with -1. After the exchange takes place, write the output of the array of each process into a unique file to make sure that the ghost points have the proper value. Assume that the domain is not periodic; that is, the top process `(rank = size - 1)` only sends and receives data from the one under it `(rank = size - 2)` and the bottom process (rank = 0) only sends and receives data from the one above it (`rank = 1`). Consider a `maxn` of 12 and use 4 processors to start with. The following illustration describes this activity.



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

Once you have completed, compiled, and executed the programme using four MPI processes, the outcome in each text file should contain the following:



Sample solution code:


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

/* This example handles a 12 x 12 mesh, on 4 MPI processes only.
*/
#define maxn 12

int main(int argc, char** argv)
{
    int rank, value, size, errcnt, toterr, i, j;
    MPI_Status status;
    double xlocal[(12/4)+2][12];
    char* pOutputFileName = (char*) malloc(20 * sizeof(char));
    FILE *pFile;

    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );

    if(size != 4) MPI_Abort(MPI_COMM_WORLD, 1);

    // Fill the data as specified
         // Fill in the process's section (middle rows) with its
own rank
    for (i=1; i<=maxn/size; i++)
```

```
        for (j=0; j<maxn; j++)
            xlocal[i][j] = rank;

    //  Fill in the ghost points with -1, ghost points are in the
first row and last row
    for (j=0; j<maxn; j++) {
        xlocal[0][j] = -1;
        xlocal[maxn/size+1][j] = -1;
    }

    // Send the last row of the middle rows to upper rank process
        // Receive a row from the lower rank process and this is
the ghostpoints in the first row
    if (rank < size - 1)
        MPI_Send(xlocal[maxn/size], maxn, MPI_DOUBLE, rank + 1,
0, MPI_COMM_WORLD);

    if (rank > 0)
        MPI_Recv(xlocal[0], maxn, MPI_DOUBLE, rank - 1, 0,
MPI_COMM_WORLD, &status);

    // Send the first row of the middle rows to lower rank process
    // Receive a row from the upper tank process and this is the
ghostpoints in the last row
    if (rank > 0)
        MPI_Send( xlocal[1], maxn, MPI_DOUBLE, rank - 1, 1,
MPI_COMM_WORLD );

    if (rank < size - 1)
        MPI_Recv( xlocal[maxn/size+1], maxn, MPI_DOUBLE, rank +
1, 1, MPI_COMM_WORLD, &status);

    /* Check that we have the correct results */
    errcnt = 0;
    for (i=1; i<=maxn/size; i++)
        for (j=0; j<maxn; j++)
            if (xlocal[i][j] != rank) errcnt++;

    for (j=0; j<maxn; j++) {
        if (xlocal[0][j] != rank - 1)
            errcnt++;
        if (rank < size-1 && xlocal[maxn/size+1][j] != rank + 1)
            errcnt++;
    }
    MPI_Reduce( &errcnt, &toterr, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD );

    // Write xlocal of each process into a unique text file (the
rank is appended into the text file name)
    snprintf(pOutputFileName, 20, "process_%d.txt", rank);
    pFile = fopen(pOutputFileName, "w");
    for (i = 0; i <= maxn/size+1; i++){
        for (j=0; j<maxn; j++){
            fprintf(pFile, "%.0f\t", xlocal[i][j]);
    }
        fprintf(pFile, "\n");
```

```
    }
    fclose(pFile);

    if (rank == 0) {
          if (toterr) printf( "! found %d errors\n", toterr );
    else
          printf( "No errors\n" );
    }

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

2. Non-Blocking Communication

In this exercise, use the non-blocking point-to-point routines instead of the blocking routines. Replace the MPI Send and MPI Recv routines in the sample solution for Question 1 with MPI Isend and MPI Irecv and use MPI Wait or MPI Waitall to test for completion of the nonblocking operations.

You may want to use these MPI routines in your solution: `MPI_Isend`, `MPI_Irecv`, `MPI_Waitall`.

3. Shifting data around

Replace the MPI Send and MPI Recv calls in your solution for Question 1 with two calls to MPI Sendrecv. The first call should shift data up; that is, it should send data to the processor above and receive data from the processor below. The second call to MPI Sendrecv should reverse this; it should send data to the processor below and receive from the processor above.

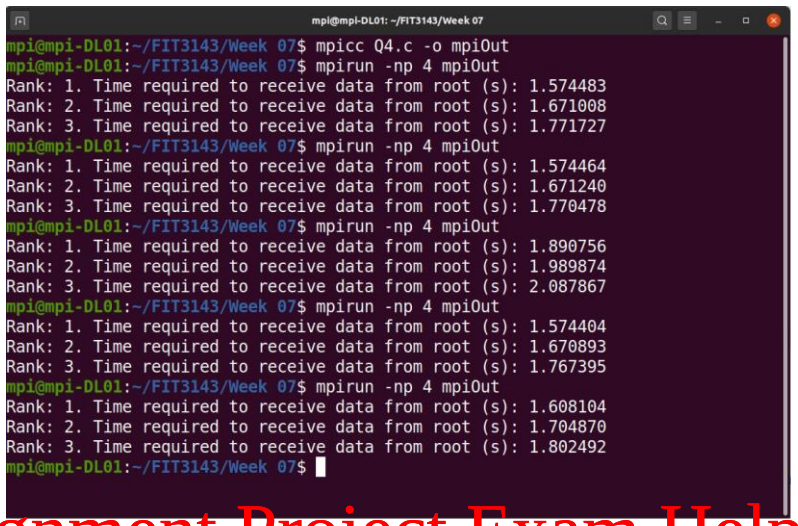
You may want to use these MPI routines in your solution: `MPI_Sendrecv`

4. Performance analysis between blocking and non-blocking send for large data transfer between MPI processes

The following code describes a send and receive operation between a root process (i.e., `my_rank == 0`) and other processes within a MPI communicator. In detail, all MPI processes create a large heap array. The root process populates its array with random values and then sends the content of its array to other processes. The non-root processes receive the data from the root process and measures the time taken to receive the data.

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

#define SIZE 100000000 // You can reduce this value based on
available system memory

int main(){
    int my_rank;
    int p;
    int source;
    int dest;
    int tag = 0;
    int *pData;
    int i;
    MPI_Status status;

    struct timespec start, end;
    double time_taken;
    int provided;

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

    pData = (int*)malloc(SIZE * sizeof(int));

    if(my_rank == 0)
    {
        srand(time(NULL));
        for(i = 0; i < SIZE; i++){
            pData[i] = rand() % 1500 + 1;
        }

        for(i = 1; i < p; i++){
            MPI_Send(pData,     SIZE,     MPI_INT,     i,     i,
MPI_COMM_WORLD);
        }
    }else{
        clock_gettime(CLOCK_MONOTONIC, &start);
        MPI_Recv(pData,     SIZE,     MPI_INT,     0,     my_rank,
MPI_COMM_WORLD, &status);
        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("Rank: %d. Time required to receive data from root
(s): %lf\n", my_rank, time_taken);
    }
    MPI_Finalize();
    free(pData);

    return 0;
}
```

The code above (or in the previous page) was compiled and executed using OpenMPI with four processes in a single computer with six CPU cores and 16 GBytes of memory. The following screenshot illustrates the terminal which displays the time required by each process to receive the data from the root process.



```
mpi@mpi-DL01: ~/FIT3143/Week 07
mpi@mpi-DL01:~/FIT3143/Week 07$ mpicc Q4.c -o mpiOut
mpi@mpi-DL01:~/FIT3143/Week 07$ mpirun -np 4 mpiOut
Rank: 1. Time required to receive data from root (s): 1.574483
Rank: 2. Time required to receive data from root (s): 1.671008
Rank: 3. Time required to receive data from root (s): 1.771727
mpi@mpi-DL01:~/FIT3143/Week 07$ mpirun -np 4 mpiOut
Rank: 1. Time required to receive data from root (s): 1.574464
Rank: 2. Time required to receive data from root (s): 1.671240
Rank: 3. Time required to receive data from root (s): 1.770478
mpi@mpi-DL01:~/FIT3143/Week 07$ mpirun -np 4 mpiOut
Rank: 1. Time required to receive data from root (s): 1.890756
Rank: 2. Time required to receive data from root (s): 1.989874
Rank: 3. Time required to receive data from root (s): 2.087867
mpi@mpi-DL01:~/FIT3143/Week 07$ mpirun -np 4 mpiOut
Rank: 1. Time required to receive data from root (s): 1.574404
Rank: 2. Time required to receive data from root (s): 1.670893
Rank: 3. Time required to receive data from root (s): 1.767395
mpi@mpi-DL01:~/FIT3143/Week 07$ mpirun -np 4 mpiOut
Rank: 1. Time required to receive data from root (s): 1.608104
Rank: 2. Time required to receive data from root (s): 1.704870
Rank: 3. Time required to receive data from root (s): 1.802492
mpi@mpi-DL01:~/FIT3143/Week 07$
```


Analysing the screenshot above, notice that each time the compiled program is executed, the second and third process ranks (i.e., Ranks 2 and 3) would require a slightly longer time to receive the data from the root process. Note that you should compile and execute the code above in your local computer or virtual machine. Results may vary when executing the compiled code above across different platform environments (e.g., Virtual machine, native Linux, MAC OS or even CAAS). Nevertheless, we should observe a longer time being required by the latter process ranks in receiving the data from the root process (especially when running the program on a cluster with lower network throughput and higher network latency).

a) Modify the code above to use non-blocking send (i.e., `MPI_Isend` & `MPI_Waitall`). Compile and execute the modified code using the same number of MPI processes (i.e., four processes) as described earlier in this task. Analyse and compare the performance when using blocking and non-blocking send operations.

b) Instead of using non-blocking send, parallelize the `for` loop which calls the `MPI_Send` function in the code above using OpenMP (i.e., Hybrid MPI and OpenMP). To do this:

   i. Change `MPI_Init` to `MPI_Init_thread`. Click here for documentation on the MPI Init thread function. Specify `MPI_THREAD_MULTIPLE` as the third argument for the `MPI_Init_thread` function.

   ii. Apply the `#pragma omp parallel for` directive above the `for` loop which calls the `MPI_Send` function. You should specify which parameters are shared or privatized. You could also specify a scheduling construct and the number of threads to call for this omp construct.

   iii. Compile the code using mpicc with the -fopenmp option.

   iv. Execute the compiled program using the same number of MPI processes (i.e., four processes) as described earlier in this task. Analyse and compare the performance when using MPI only and Hybrid MPI + OpenMP for the send operation.

Note: Executing the compiled code on CAAS is optional for this lab. However, we do encourage you to use CAAS especially if you are facing hurdles executing the compiled code in your virtual machine due to limited CPU or memory. The CAAS documentation slides includes an example (Slide #23) on submitting a job script for MPI + OpenMP for your reference.