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

- The purpose of this lab is to explore parallel data structures and non-blocking communication in 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:

- Practice parallel algorithm design and development
- Practice non-blocking communication 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)

Task 1: A Parallel Data Structure

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).

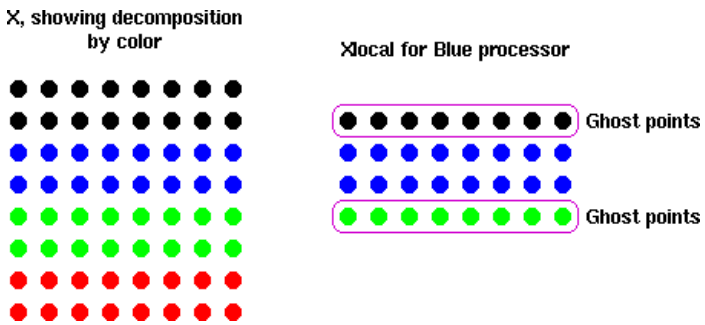
If that was all that there was to it, there wouldn't be anything to do. However, for the computation that we're going to perform on this data structure, we'll need the adjacent values. That is, to compute a new `x[i][j]`, we will need

```
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 a parallel code using MPI 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, test 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.



In this exercise (i.e., Task 1), **use non-blocking MPI routines** instead of the blocking routines you have learned before. Use 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.

Exchanging data with MPI_Sendrecv (worked example)

For Question 1, we can use MPI Sendrecv to exchange data with the neighboring processors. That is, processors 0 and 1 exchange, 2 and 3 exchange, etc. Then 1 and 2 exchange, 3 and 4, etc. This "head-to-head" exchange may be more efficient on some systems.

Sample Solution:

```
#include  <stdio.h>
#include  "mpi.h"
/*  This  example  handles  a  12  x  12  mesh,  on  4  proce
ssors  only.  */
#define  maxn  12
int  main(  argc,  argv  )
int  argc;
char  **argv;
{
       int  rank,  value,  size,  errcnt,  toterr,  i,  j;
       int  up_nbr,  down_nbr;
       MPI_Status  status;
       double  x[12][12];
       double  xlocal[(12/4)+2][12];
       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  );
       /*  xlocal[][0]  is  lower  ghostpoints,  xlocal[][max
n+2]  is  upper  */
       /*  Fill  the  data  as  specified  */
       for  (i=1;  i<=maxn/size;  i++)
for  (j=0;  j<maxn;  j++)
       xlocal[i][j]  =  rank;
       for  (j=0;  j<maxn;  j++)  {
xlocal[0][j]  =  --1;
xlocal[maxn/size+1][j]  =  --1;
       }
       /*  Send  up  and  receive  from  below  (shift  up)
*/
       /*  Note  the  use  of  xlocal[i]  for  &xlocal[i][0]
 */
       /*  Note  that  we  use  MPI_PROC_NULL  to  remove  th
e  if  statements  that
             would  be  needed  without  MPI_PROC_NULL  */
       up_nbr  =  rank  +  1;
       if  (up_nbr  >=  size)  up_nbr  =  MPI_PROC_NULL;
```

```
        down_nbr  =  rank  --  1;
        if  (down_nbr  <  0)  down_nbr  =  MPI_PROC_NULL;
        MPI_Sendrecv(  xlocal[maxn/size],  maxn,  MPI_DOUBLE,
 up_nbr,  0,
   xlocal[0],  maxn,  MPI_DOUBLE,  down_nbr,  0,
   MPI_COMM_WORLD,  &status  );
        /*  Send  down  and  receive  from  above  (shift  dow
n)  */
        MPI_Sendrecv(  xlocal[1],  maxn,  MPI_DOUBLE,  down_nb
r,  1,
   xlocal[maxn/size+1],  maxn,  MPI_DOUBLE,  up_nbr,  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  );
        if  (rank  ==  0)  {
if  (toterr)
        printf(  "!  found  %d  errors\n",  toterr  );
else
        printf(  "No  errors\n"  );
        }
        MPI_Finalize(  );
        return 0;
}
```

Task 2: Shifting data around

Your task is to 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.

Compare Task 1 approach with Task 2 approach. Which approach is efficient and why?

MONASH University

OPTIONAL BONUS ACTIVITIES (2 MARKS)

1. Parallel Matrix Multiplication using Non-Blocking Communication

For matrix multiplication, the product of an $m \times p$ of ***matrix a*** with a $p \times n$ of ***matrix b*** results in an $m \times n$ matrix denoted as ***c*** as shown below:

$$c_{i,j} = \sum_{k=0}^{p-1} a_{ik} b_{kj}; \quad \begin{matrix} i = 0,1,2,\ldots,m-1 \\ j = 0,1,2,\ldots,n-1 \end{matrix}$$

Using the equation above, write a parallel matrix multiplication code using MPI to multiply large matrices. You can extend the sample code on vector product which was discussed in the pre-lab activities to implement a matrix multiplication. More importantly, demonstrate the use of non-blocking communication in your code.

Measure the actual speed up.