MANIPAL INSTITUTE OF TECHNOLOGY
Manipal – 576 104

DEPARTMENT OF COMPUTER SCIENCE & ENGG.

MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

CERTIFICATE

This is to certify that Ms./Mr. ……..…… Reg. No. …..…… Section: …… Roll No: ……... has satisfactorily completed the lab exercises prescribed for Parallel Programming Lab [CSE 3263] of Third Year B. Tech. Degree at MIT, Manipal, in the academic year 2022-2023.

Date: ……..……..

Signature
Faculty in Charge

Signature
Head of the Department

CONTENTS

LAB NO.	TITLE	PAGE NO.	REMARKS
	Course Objectives and Outcomes	i	
	Evaluation plan	i	
	Instructions to the Students	ii	
1	Introduction to execution environment of MPI	1	
2	Point to Point Communications in MPI	5	
3	Collective communications in MPI	9	
4	Collective communications and Error handling in MPI	13	
5	OpenCL introduction and programs on vectors	17	
6	OpenCL programs on strings,sorting and to check the execution time in OpenCL	30	
7	Programs on Arrays in CUDA	35	
8	Programs on Strings in CUDA	42	
9	Programs on Matrix in CUDA	47	
10	Programs on Matrix in CUDA	50	
11	Programs on CUDA Device memory types and synchronization	52	
12	Programs on image processing applications in CUDA	57	
13	References	63	

Course Objectives

- Learn different APIs used in MPI for point to point, collective communications and error handling
- Learn how to write host and kernel code for device neutral architecture using OpenCL
- Learn how to write host and kernel code in CUDA for nVIDIA GPU card
- To develop the skills of design and implement parallel algorithms using different parallel programming environment

Course Outcomes

At the end of this course, students will be able to

- Write MPI programs using point-to-point and collective communication primitives.
- Solve and test OpenCL programs using GPU architecture
- Develop CUDA programs for parallel applications

Evaluation plan

- Internal Assessment Marks : 60%
  - ➢ Continuous Evaluation : 60%

    Continuous evaluation component (for each evaluation):10 marks

    The assessment will depend on punctuality, program execution, maintaining the observation note and answering the questions in viva voce.

- End semester assessment of 2 hour duration: 40 %

INSTRUCTIONS TO THE STUDENTS

Pre- Lab Session Instructions

1. Students should carry the Lab Manual Book and the required stationery to every lab session.
2. Be in time and follow the institution dress code.
3. Must Sign in the log register provided.
4. Make sure to occupy the allotted seat and answer the attendance
5. Adhere to the rules and maintain the decorum.
6. Students must come prepared for the lab in advance.

In- Lab Session Instructions

- Follow the instructions on the allotted exercises.
- Show the program and results to the instructors on completion of experiments.
- On receiving approval from the instructor, copy the program and results in the Lab record
- Prescribed textbooks and class notes can be kept ready for reference if required.

General Instructions for the exercises in Lab

- Implement the given exercise individually and not in a group.
- Observation book should be complete with program, proper input output clearly showing the parallel execution in each process. Plagiarism (copying from others) is strictly prohibited and would invite severe penalty in evaluation.
- The exercises for each week are divided under three sets:
  - Solved example
  - Lab exercises - to be completed during lab hours
  - Additional Exercises - to be completed outside the lab or in the lab to enhance the skill
- In case a student misses a lab class, he/ she must ensure that the experiment is completed during the repetition class with the permission of the faculty concerned but credit will be given only to one day's experiment(s).
- Questions for lab tests and examination are not necessarily limited to the questions in the manual, but may involve some variations and / or combinations of the questions.

THE STUDENTS SHOULD NOT

- Bring mobile phones or any other electronic gadgets to the lab.
- Go out of the lab without permission.

Lab No 1: **Date:**

Introduction to execution environment of MPI

Objectives:

In this lab, student will be able to

1. Understand the execution environment of MPI programs
2. Learn the various concept of parallel programming
3. Learn and use the Basics API available in MPI

I.Introduction

In order to reduce the execution time work is carried out in parallel. Two types of parallel programming are:

- Explicit parallel programming
- Implicit parallel programming

Explicit parallel programming – These are languages where the user has full control and has to explicitly provide all the details. Compiler effort is minimal.

Implicit parallel programming – These are sequential languages where the compiler has full responsibility for extracting the parallelism in the program.

Parallel Programming Models:

- Message Passing Programming
- Shared Memory Programming

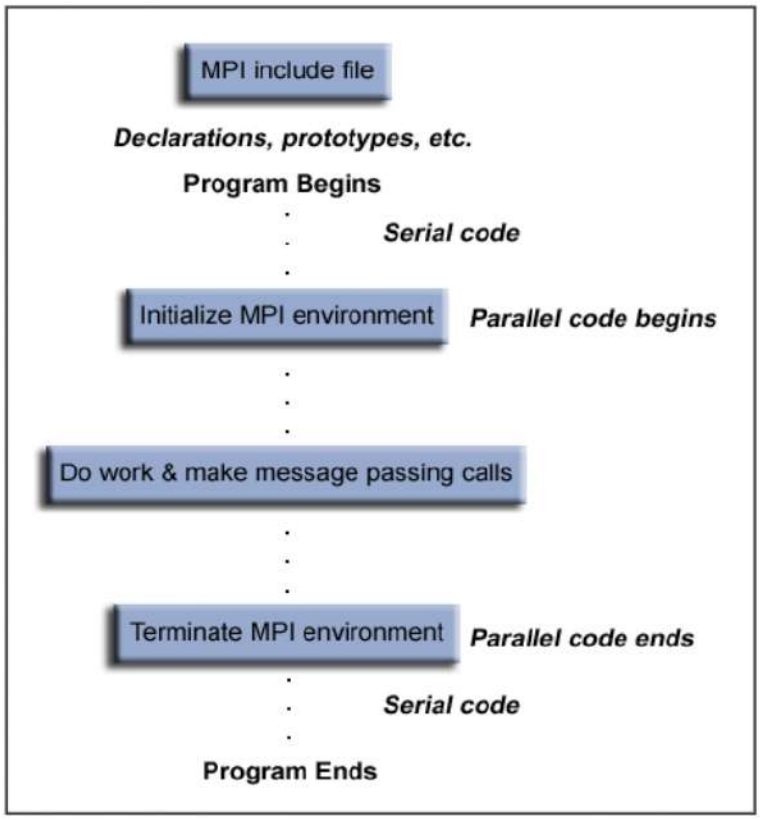
Message Passing Programming:

- In message passing programming, programmers view their programs (Applications) as a collection of co-operating processes with private (local) variables.
- The only way for an application to share data among processors is for programmer to explicitly code commands to move data from one processor to another.

Message Passing Libraries: There are two message passing libraries available. They are:

- PVM – Parallel Virtual Machine
- MPI – Message Passing Interface. It is a set of parallel APIs which can be used with languages such as C and FORTRAN.

II. MPI Program Structure:



Communicators and Groups:

- MPI assumes static processes.
- All the processes are created when the program is loaded.
- No process can be created or terminated in the middle of program execution.
- There is a default process group consisting of all such processes identified by **MPI_COMM_WORLD**.

III. MPI Environment Management Routines:

MPI_Init: Initializes the MPI execution environment. This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program.

```
MPI_Init (&argc,&argv);
```

MPI_Comm_size: Returns the total number of MPI processes to the variable size in the specified communicator, such as MPI_COMM_WORLD.

```
MPI_Comm_size(Comm,&size);
```

MPI_Comm_rank: Returns the rank of the calling MPI process within the specified communicator. Each process will be assigned a unique integer rank between 0 and size - 1 within the communicator MPI_COMM_WORLD. This rank is often referred to as a process ID.

```
MPI_Comm_rank Comm,&rank);
```

MPI_Finalize: Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program. No other MPI routines may be called after it.

```
MPI_Finalize ();
```

Solved Example:

Write a program in MPI to print total number of process and rank of each process.

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
        int rank,size;

        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD,&rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        printf("My rank is %d in total %d process",rank,size);
        MPI_Finalize();
        return 0;
}
```

Steps to execute a MPI program is provided in the form of video which is available in individual systems.

Lab Exercises:

1. Write a simple C++ program to multiply two matrices of size M*x*N.
2. Write a program in MPI where even ranked process prints Hello and odd ranked process prints World.

3. Write a program in MPI to simulate simple calculator. Perform each operation using different process in parallel.
4. Write a C++ program to accept the input from .txt file and extract the text enclosed <TEXT> </TEXT> tag. Display the occurrence of each string present in the text tag.
   File : https://drive.google.com/open?id=1aJMSrpJPFD1czy5KCAuaa7cpatNYkPwd

Additional Exercises:

1. Write a program in C++ to count the words in a file and sort it in descending order of frequency of words i.e. highest occurring word must come first and least occurring word must come last.
2. Write a MPI program to find the prime numbers between 1 and 100 using two processes.

Lab No 2: **Date:**

Point to Point Communications in MPI

Objectives:

In this lab, student will be able to

1. Understand the different APIs used for point to point communication in MPI
2. Learn the different modes available in case of blocking send operation

Point to Point communication in MPI

- MPI point-to-point operations typically involve message passing between two, and only two, different MPI tasks. One task is performing a send operation and the other task is performing a matching receive operation.
- MPI provides both blocking and non-blocking send and receive operations.

Sending message in MPI

- **Blocked Send** sends a message to another processor and waits until the receiver has received it before continuing the process. Also called as **Synchronous send**.
- **Send** sends a message and continues without waiting. Also called as **Asynchronous send.**

There are multiple communication modes used in blocking send operation:

- **Standard mode**
- **Synchronous mode**
- **Buffered mode**

Standard mode

This mode blocks until the message is buffered.

```
MPI_Send(&Msg, Count, Datatype, Destination, Tag, Comm);
```

- First 3 parameters together constitute message buffer. The **Msg** could be any address in sender's address space. The **Count** indicates the number of data elements of a particular type to be sent. The **Datatype** specifies the message type. Some Data types available in MPI are: MPI_INT, MPI_FLOAT, MPI_CHAR, MPI_DOUBLE, MPI_LONG
- Next 3 parameters specify message envelope. The **Destination** specifies the rank of the process to which the message is to be sent.
- **Tag:** The **tag** is an integer used by the programmer to label different types of messages and to restrict message reception.

- **Communicator:** Major problem with tags is that they are specified by users who can make mistakes. **Context** are allocated at run time by the system in response to user request and are used for matching messages. The notions of **context and group** are combined in a single object called a communicator (**Comm**).
- The default process group is **MPI_COMM_WORLD**.

Synchronous mode

This mode requires a send to block until the corresponding receive has occurred.

```
MPI_Ssend(&Msg, Count, Datatype, Destination, Tag,  Comm);
```

Buffered mode

```
MPI_Bsend(&Msg, Count, Datatype, Destination, Tag,  Comm);
```

In this mode a send assumes availability of a certain amount of buffer space, which must be previously specified by the user program through a routine call that allocates a user buffer.

```
MPI-Buffer_attach(buffer, size);
```

This buffer can be released by

```
MPI_Buffer_detach(*buffer, *size);
```

Receiving message in MPI

```
MPI_Recv(&Msg, Count, Datatype, Source, Tag, Comm, &status);
```

- Receive a message and block until the requested data is available in the application buffer in the receiving task.
- The **Msg** could be any address in receiver's address space. The **Count** specifies number of data items. The **Datatype** specifies the message type. The **Source** specifies the rank of the process which has sent the message. The **Tag** and **Comm** should be same as that is used in corresponding send operation. The status is a structure of type status which contains following information: Sender's rank, Sender's tag and number of items received

Finding execution time in MPI

MPI_Wtime: Returns an elapsed wall clock time in seconds (double precision) on the calling processor.

- **MPI_Wtime ()**

Solved Example:

Write a MPI program using standard send. The sender process sends a number to the receiver. The second process receives the number and prints it.

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
        int rank,size,x;
        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD,&rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Status status;
        if(rank==0)
         {
          Printf("Enter a value in master process:");
          scanf("%d",&x);
          MPI_Send(&x,1,MPI_INT,1,1,MPI_COMM_WORLD);
          fprintf(stdout,"I have send %d from process 0\n",x);
          fflush(stdout);
         }
        else
        {
         MPI_Recv(&x,1,MPI_INT,0,1,MPI_COMM_WORLD,&status);
         fprintf(stdout,"I have received %d in process 1\n",x);
         fflush(stdout);
         }
       MPI_Finalize();
       return 0;
}
```

Lab Exercises:

1) Write a MPI program using synchronous send. The sender process sends a word to the receiver. The second process receives the word, toggles each letter of the word and sends it back to the first process. Both process use synchronous send operations.
2) Write a MPI program where the master process (process 0) sends a number to each of the slaves and the slave processes receives the number and prints it. Use standard send.
3) Write a MPI program to read N elements of the array in the root process (process 0) where N is equal to the total number of process. The root process sends one value to each of the

slaves. Let even ranked process find square of the received element and odd ranked process find cube of received element. Use Buffered send.

4) Write a MPI program to read an integer value in the root process. Root process sends this value to process1, Process1 sends this value to Process2 and so on. Last process sends the value back to root process. When sending the value each process will first increment the received value by one. Write the program using point to point communication routines.

5) Write a MPI program to compute standard deviation of N random numbers using collective communication APIs. Assume N is evenly divisible by number of processes.
Formula :

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

Example :
N = 20
N Random No = 9, 2, 5, 4, 12, 7, 8, 11, 9, 3, 7, 4, 12, 5, 4, 10, 9, 6, 9, 4
SD = 2.983

7) Write a MPI program to accept the input from .txt file by root process, and extract the text enclosed <TEXT> </TEXT> tag. Send each statement to seperate process and count the total number of word by each process. Display the word count by each process and send back the count to root process. Finally display the total word count by the root process.

Additional Exercises:

1) Write a MPI program to read N elements of an array in the root. Search a number in this array using root and another process. Print the result in the root.
2) Write a MPI program to read N elements of an array in the master process. Let N process including master process check the array values are prime or not.
3) Write a MPI program to read value of N in the root process. Using N processes including root find out 1!+(1+2)+3!+(1+2+3+4)+5!+(1+2+3+4+5+6)+……..+n! or (1+2+…..+n) depending on whether n is odd or even and print the result in the root process.

Lab No 3: **Date:**

Collective Communications in MPI

Objectives:

In this lab, student will be able to

1. Understand the usage of collective communication in MPI
2. Learn how to broadcast messages from root
3. Learn and use the APIs for distributing values from root and gathering the values in the root

Collective Communication routines

When **all processes** in a group participate in a global communication operation, the resulting communication is called a **collective communication**.

MPI_Bcast:

```
MPI_Bcast (Address, Count, Datatype, Root, Comm);
```

The process ranked **Root** sends the same message whose content is identified by the triple (Address,Count,Datatype) to all processes(including itself) in the communicator **Comm**.

MPI_Scatter:

```
MPI_Scatter( SendBuff, Sendcount, SendDatatype, RecvBuff, Recvcount,
             RecvDatatype, Root, Comm);
```

Ensures that the **Root** process sends out personalized messages, which are in rank order in its send buffer, to all the N processes (including itself).

MPI_Gather:

```
MPI_Gather( SendAddress, Sendcount, SendDatatype, RecvAddress, RecvCount,
            RecvDatatype, Root, Comm);
```

The **root** process receives a personalized message from all N processes. These N received messages are concatenated in rank order and stored in the receive buffer of the root process.

Total Exchange:

In routine **MPI_Alltoall()** each process sends a personalized message to every other process including itself. This operation is equivalent to N gathers, each by a different process and in all N^2 messages are exchanged.

Solved Example:

Write a MPI program to read N values of the array in the root process. Distribute these N values among N processes. Every process finds the square of the value it received. Let every process return these value to the root and root process gathers and prints the result. Use collective communication routines.

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


int main(int argc, char *argv[])
{
        int rank,size,N,A[10],B[10], c, i;

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

        if(rank==0)
        {
            N=size;
            fprintf(stdout,"Enter  %d values:\n",N);
            fflush(stdout);
            for(i=0; i<N; i++)
               scanf("%d",&A[i]);
         }
       MPI_Scatter(A,1,MPI_INT,&c,1,MPI_INT,0,MPI_COMM_WORLD);
       fprintf(stdout,"I have received %d in process %d\n",c,rank);
       fflush(stdout);

       c=c*c;
       MPI_Gather(&c,1,MPI_INT,B,1,MPI_INT,0,MPI_COMM_WORLD);

       if(rank==0)
        {
           fprintf(stdout,"The Result gathered in the root \n");
           fflush(stdout);
           for(i=0; i<N; i++)
             fprintf(stdout,"%d \t",B[i]);
           fflush(stdout);
```

```
        }

        MPI_Finalize();
        return 0;
}
```

Lab Exercises:

1) Write a MPI program to read N values in the root process. Root process sends one value to each process. Every process receives it and finds the factorial of that number and returns it to the root process. Root process gathers the factorial and finds sum of it. Use N number of processes.
2) Write a MPI program to read a value M and N*x*M elements in the root process. Root process sends M elements to each process. Each process finds average of M elements it received and sends these average values to root. Root collects all the values and finds the total average. Use collective communication routines. Use N number of processes.
3) Write a MPI program to read a string. Using N processes (string length is evenly divisible by N), find the number of non-vowels in the string. In the root process print number of non-vowels found by each process and print the total number of non-vowels.
4) Write a MPI Program to read two strings S1 and S2 of same length in the root process. Using *N* process including the root (string length is evenly divisible by N), produce the resultant string as shown below. Display the resultant string in the root process. Write the program using Collective communication routines.
   Example:

   String S1: string String S2: length Resultant String : slternigntgh

5) Write a MPI program using collective communication, to replace all even elements of array A to 1 and replace all odd elements to 0 of size N. Display the resultant array A, count of all even and odd numbers in root process. Assume N is evenly divisible by number of processes.
   Example :
   Input Array (A): 1 2 3 4 5 6 7 8 9
   Resultant Array (A): 0 1 0 1 0 1 0 1 0
   Even (Count) = 4
   Odd (Count) = 5
6) Without using Point to Point communication routines write a program in MPI to perform the following in parallel using 4 Processes. The output array elements are as shown in the example are to be displayed by the root.

   Sample Input Output:
   Input Array: 1 2 3 5 0 3 8 7 2 0 1 7 2 3 0 9
   Output Array: 1 3 6 11 0 3 11 18 2 2 3 10 2 5 5 14

Additional Exercises:

1) Write a MPI Program to read a string of length M in the root process. Using N processes (N evenly divides M) including the root toggle the characters and find the ASCII values of these toggled characters. Display the toggled characters and ASCII values in the root process.
2) Write a program to read a value M and N*x*M number of elements in the root. Using N processes do the following task. Find the square of first M numbers, Find the cube of next M numbers and so on. Print the results in the root.
3) Write a program to read a value M and N*x*M number of elements in the root. Using N number of processes find the sum 1+2…..+array element of each element and print the result in the root.

I/p:	M=2			N=3		
Array:	2	4	3	2	5	3
Result:	3	10	6	3	15	6

Lab No 4: **Date:**

Collective Communications and Error Handling in MPI

Objectives:

In this lab, student will be able to

1. Understand the different aggregate functions used in MPI
2. Learn how to write MPI programs using both point to point and collective communication routines
3. Learn and use the APIs for handling errors in MPI

I.Aggregation Functions

MPI provides two forms of aggregation

- **Reduction**
- **Scan**

Reduction:

```
MPI_Reduce (SendAddress, RecvAddress, Count, Datatype, Op, Root, Comm);
```

This routine reduces the partial values stored in **SendAddress** of each process into a final result and stores it in **RecvAddress** of the **Root** process. The reduction operator is specified by the **Op** field. Some of the reduction operator available in MPI are: MPI_SUM, MPI_MAX, MPI_MIN, MPI_PROD

Scan:

```
MPI_Scan (SendAddress, RecvAddress, Count, Datatype, Op, Comm);
```

This routine combines the partial values into N final results which it stores in the **RecvAddress** of the N processes. Note that root field is absent here. The scan operator is specified by the **Op** field. Some of the scan operator available in MPI are: MPI_SUM, MPI_MAX, MPI_MIN, MPI_PROD

MPI_Barrier(Comm) :This routine synchronizes all processes in the communicator **Comm**. They wait until all N processes execute their respective MPI_Barrier.

Note: All collective communication routines except MPI_Barrier, employ a standard blocking mode of point-to-point communication.

Error Handling in MPI:

- An MPI *communicator* is more than just a group of process that belong to it. Amongst the items that the communicator hides inside is an *error handler*. The error handler is called every time an MPI error is detected within the communicator.
- The predefined default error handler, which is called **MPI_ERRORS_ARE_FATAL**, for a newly created communicator or for MPI_COMM_WORLD is to *abort the whole parallel program* as soon as any MPI error is detected. There is another predefined error handler, which is called **MPI_ERRORS_RETURN**.
- The default error handler can be replaced with this one by calling function **MPI_Errhandler_set**, for example:

```
MPI_Errhandler_set(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
```

- The only **error code** that MPI standard itself defines is **MPI_SUCCESS**, i.e., no error. But the meaning of an error code can be extracted by calling function **MPI_Error_string**. On top of the above MPI standard defines the so called ***error classes***. The **error class** for a given error code can be obtained by calling function **MPI_Error_class**.
- Error classes can be converted to comprehensible error messages by calling the same function that does it for error codes, i.e., **MPI_Error_string**. The reason for this is that error classes are implemented as a subset of error codes.

Solved Example:

Write a MPI program using N processes to find 1! + 2! +…..+N!. Use collective communication routines.

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char* argv[])
{
     int rank,size,fact=1, factsum, i;

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

      for(i=1; i<=rank+1; i++)
          fact = fact * i;

     MPI_Reduce (&fact,&factsum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

      if(rank==0)
```

```
        printf("Sum of all the factorial=%d",factsum);

    MPI_Finalize();
    exit(0);
}
```

Lab Exercises:

1) Write a MPI program using N processes to find 1! + 2! +…..+N!. Use scan.
2) Write a MPI program to calculate π-value by integrating $f(x) = 4/(1+x^2)$. Area under the curve is divided into rectangles and the rectangles are distributed to the processors.
3) Write a MPI program to read a 3 *x* 3 matrix. Enter an element to be searched in the root process. Find the number of occurrences of this element in the matrix using three processes.
4) Write a MPI program to handle different errors using error handling routines.
5) Write a MPI program to read 4 *x* 4 matrix display the following output using four processes

I/p matrix:				O/p matrix:			
1	2	3	4	1	2	3	4
1	2	3	1	2	4	6	5
1	1	1	1	3	5	7	6
2	1	2	1	5	6	9	7

Additional Exercises:

1) Write a MPI Program to read two 3 *x* 3 matrix in the root process. Using three processes including the root add the corresponding elements of A & B matrix. Display the added resultant matrix in the root.
2) Write a MPI program using collective communication that displays the sum of all prime numbers of 3 *x* 3 matrix in the root process. Use three processes.
3) Write a MPI Program to read a 3 *x* 3 matrix in the root process. Using three processes including the root find the maximum in each row and minimum in each column of this matrix. Display the maximum and minimum values in the root. Use collective communication routines.
4) Write a MPI program to read a word of length N. Using N processes including the root get output word with the pattern as shown in example. Display the resultant output word in the root.
   Example: Input : PCAP Output : PCCAAAPPPP

5) Write a MPI program to read matrix A of size 5x5. It produces a resultant matrix B of size 5x5. It sets all the principal diagonal elements of B matrix with 0. It replaces each row elements in the B matrix following manner: If the element is below the principal diagonal it replaces it with the maximum value of the column in the A matrix having the same row number of B. If the element is above the principal diagonal it replaces it with the minimum value of the column in the A matrix having the same row number of B. Produce the B Matrix using 5 processes. Use only Collective communication routines except broadcast routine.

Example:

A				
1	2	3	4	5
5	4	3	2	1
10	3	13	14	15
11	22	11	33	44
1	12	5	4	6

B				
0	**1**	**1**	1	**1**
22	**0**	**2**	**2**	**2**
13	**13**	**0**	**3**	**3**
33	**33**	**33**	**0**	**2**
44	**44**	**44**	**44**	**0**