PARALLEL & DISTRIBUTED COMPUTING LAB

(CSC17202)

LAB MANUAL

VII SEMESTER

Department of Computer Science and Engineering
INDIAN INSTITUTE OF TECHNOLOGY (INDIAN SCHOOL OF MINES)
DHANBAD – 826004, INDIA

LIST OF EXPERIMENTS

S. NO	Name of the Experiment	Page No.
	MPI PROGRAMMING	
1.	Write a program Create, compile and run an MPI "Hello world" program.	
2.	Write a program to sum of an array using MPI.	
3.	Write a program to calculate the dot product:	
4.	Write a program to calculate the definite integral of a nonnegative function using Simpson's 1/3 rule.	
5.	Write a program to compute function using the Lagrange's interpolation.	
6.	Write a program for the prefix computation of given data.	
7.	Write a program to implement the following matrix-matrix multiplication.	
8.	Write a program to compute the polynomial using shuffle and exchange network.	
9.	Write a program to implement the following matrix-vector multiplication.	
10.	Write a program to calculate the definite integral of a nonnegative function f (x) using trapezoidal rule with broadcast and reduce functions for message sharing among PEs.	
11.	Write a program to implement matrix-matrix multiplication on a 2D wrap around mesh network.	
12.	Write a program to implement odd-even transposition sort on a linear array.	
	CUDA PROGRAMMING	
1.	WAP for "Hello world" using CUDA API and compile and run over remote access on server.	
2.	Write a parallel program for vector addition using CUDA API and compile and run over remote access login on server.	
3.	Program that adds two numbers together with modified kernel Using CUDA API and compile and run over remote access login on server.	
4.	Program that adds two numbers together using a kernel function grid that is the same dimensions as the 2D array. Using CUDA API and compile and run over remote access login on server.	
5.	Define data structure for a parallel program for Matrix addition using CUDA.	
6.	Write a parallel program for Matrix addition using CUDA API and compile and run over remote access login on server.	
7.	Define data structure for a parallel program for Matrix multiplication using CUDA.	
8.	Write a parallel program for Matrix multiplication using CUDA API and compile and run over remote access login on server.	

EXPERIMENT NO: 1

Aim: - Create, compile and run an MPI "Hello world" program.
Software Required: MPI Library, GCC Compiler, Linux operating system, Personal Computer.

Program:

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define  MASTER              0

int main (int argc, char *argv[])
{
int   numtasks, taskid, len;
char hostname[MPI_MAX_PROCESSOR_NAME];

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
MPI_Get_processor_name(hostname, &len);
printf ("Hello from task %d on %s!\n", taskid, hostname);
if (taskid == MASTER)
  printf("MASTER: Number of MPI tasks is: %d\n",numtasks);
MPI_Finalize();

}
```

Discussion:

Create: Using your favorite text editor (like vi/vim)open a new file - call it whatever you'd like. It should do the following:

- Use the appropriate MPI include file
- Identify task 0 as the "master" task
- Initialize the MPI environment
- Get the total number of tasks
- Get the task's rank (who it is)
- Get the name of the processor it is executing on
- Print a hello message that includes its task rank and processor name
- Have the master task alone print the total number of tasks
- Terminate the MPI environment

- **Compile:** Use a C or Fortran MPI compiler command. For example:
- **mpicc -w -o hello myhello.c**

myhello.c represent your source file - use your actual source file name
The **-o** compiler flag specifies the name for your executable
The **-w** compiler flag is simply being used to suppress annoying warning messages.

When you get a clean compile, proceed.

- **Run:** Use the **mpiexec** command to run your MPI executable. Be sure to use n number of tasks on different nodes. For example:

mpiexec –n <number of tasks> hello

The arguments, argc and argv, are pointers to the arguments to main, argc, and argv. However, when our program doesn't use these arguments, we can just pass NULL for both. Like most MPI functions, MPI Init returns an int error code, and in most cases we'll ignore these error codes.
MPI Finalize tells the MPI system that we're done using MPI, and that any resources allocated for MPI can be freed.
In MPI a communicator is a collection of processes that can send messages to each other. One of the purposes of MPI_Init is to define a communicator that consists of all of the processes started by the user when she started the program. This communicator is called MPI_COMM_WORLD.

EXPERIMENT NO: 2

Aim **Write a MPI program to sum of an array .**

Software Required: - MPI Library, GCC Compiler, Linux operating system, Personal Computer.

Program:

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

// size of array
#define n 10

int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// Temporary array for slave process
int a2[1000];

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

	int pid, np,
		elements_per_process,
		n_elements_recieved;
	// np -> no. of processes
	// pid -> process id

	MPI_Status status;

	// Creation of parallel processes
	MPI_Init(&argc, &argv);

	// find out process ID,
	// and how many processes were started
	MPI_Comm_rank(MPI_COMM_WORLD, &pid);
	MPI_Comm_size(MPI_COMM_WORLD, &np);

	// master process
	if (pid == 0) {
		int index, i;
		elements_per_process = n / np;

		// check if more than 1 processes are run
		if (np > 1) {
			// distributes the portion of array
			// to child processes to calculate
			// their partial sums
			for (i = 1; i < np - 1; i++) {
				index = i * elements_per_process;

				MPI_Send(&elements_per_process,
					1, MPI_INT, i, 0,
```

```
                                        MPI_COMM_WORLD);
                        MPI_Send(&a[index],
                                        elements_per_process,
                                        MPI_INT, i, 0,
                                        MPI_COMM_WORLD);
                }

                // last process adds remaining elements
                index = i * elements_per_process;
                int elements_left = n - index;

                MPI_Send(&elements_left,
                                1, MPI_INT,
                                i, 0,
                                MPI_COMM_WORLD);
                MPI_Send(&a[index],
                                elements_left,
                                MPI_INT, i, 0,
                                MPI_COMM_WORLD);
        }

        // master process add its own sub array
        int sum = 0;
        for (i = 0; i < elements_per_process; i++)
                sum += a[i];

        // collects partial sums from other processes
        int tmp;
        for (i = 1; i < np; i++) {
                MPI_Recv(&tmp, 1, MPI_INT,
                                MPI_ANY_SOURCE, 0,
                                MPI_COMM_WORLD,
                                &status);
                int sender = status.MPI_SOURCE;

                sum += tmp;
        }

        // prints the final sum of array
        printf("Sum of array is : %d\n", sum);
}
// slave processes
else {
        MPI_Recv(&n_elements_recieved,
                        1, MPI_INT, 0, 0,
                        MPI_COMM_WORLD,
                        &status);

        // stores the received array segment
        // in local array a2
        MPI_Recv(&a2, n_elements_recieved,
                        MPI_INT, 0, 0,
                        MPI_COMM_WORLD,
                        &status);

        // calculates its partial sum
        int partial_sum = 0;
```

```
                for (int i = 0; i < n_elements_recieved; i++)
                        partial_sum += a2[i];

                // sends the partial sum to the root process
                MPI_Send(&partial_sum, 1, MPI_INT,
                                        0, 0, MPI_COMM_WORLD);
        }

        // cleans up all MPI state before exit of process
        MPI_Finalize();

        return 0;
}
```

Discussion: -

Message Passing Interface (MPI) is a library of routines that can be used to create parallel programs in C . It allows users to build parallel applications by creating parallel processes and exchange information among these processes.

MPI uses two basic communication routines:

- **MPI_Send**, to send a message to another process.
- **MPI_Recv**, to receive a message from another process.

To reduce the time complexity of the program, parallel execution of sub-arrays is done by parallel processes running to calculate their partial sums and then finally, the master process(root process) calculates the sum of these partial sums to return the total sum of the array.

Exercise

1. Write an MPI program that computes a tree-structured global sum. First write your program for the special case in which comm sz is a power of two. Then, after you've gotten this version working, modify your program so that it can handle any comm_sz.
2. Write an MPI program that computes a global sum using a butterfly. First write your program for the special case in which comm_sz is a power of two. Can you modify your program so that it will handle any number of processes?

EXPERIMENT NO: 3

Aim: - Write a program to calculate the following dot product:

$$x.y = x_0 y_0 + x_1 y_1 + \square + x_{n-1} y_{n-1}$$

Software Required: - MPI Library, GCC Compiler, Linux operating system, Personal Computer.

Program:

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

/* Define length of dot product vectors */
#define VECLEN 100

int main (int argc, char* argv[])
{
int i,myid, numprocs, len=VECLEN;
double *x, *y;
double mysum, allsum;

/* MPI Initialization */
MPI_Init (&argc, &argv);
MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank (MPI_COMM_WORLD, &myid);

/*
  Each MPI task performs the dot product, obtains its partial sum, and then calls
  MPI_Reduce to obtain the global sum.
*/
if (myid == 0)
 printf("Starting omp_dotprod_mpi. Using %d tasks...\n",numprocs);

/* Assign storage for dot product vectors */
x = (double*) malloc (len*sizeof(double));
y = (double*) malloc (len*sizeof(double));

/* Initialize dot product vectors */
for (i=0; i<len; i++) {
 x[i]=1.0;
 y[i]=x[i];
 }

/* Perform the dot product */
mysum = 0.0;
for (i=0; i<len; i++)
 {
  mysum += x[i] * y[i];
 }

printf("Task %d partial sum = %f\n",myid, mysum);

/* After the dot product, perform a summation of results on each node */
```

```
MPI_Reduce (&mysum, &allsum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
if (myid == 0)
  printf ("Done. MPI version: global sum  =  %f \n", allsum);

free (x);
free (y);
MPI_Finalize();
}
```

Discussion:

MPI_Comm_size returns in its second argument the number of processes in the communicator, and MPI_Comm_rank returns in its second argument the calling process' rank in the communicator. MPI_Reduce stores the result of a global operation (e.g., a global sum) on a single designated process

Exercise

1.Write an MPI program that implements multiplication of a vector by a scalar and dot product. The user should enter two vectors and a scalar, all of which are read in by process 0 and distributed among the processes. The results are calculated and collected onto process 0, which prints them. You can assume that n, the order of the vectors, is evenly divisible by comm_sz.

EXPERIMENT NO: 4

Aim: - Write a program to calculate the definite integral of a nonnegative function f (x) using Simpson's 1/3 rule:

$$I = \int_a^b f(x)dx$$

Software Required: - MPI Library, GCC Compiler, Linux operating system, Personal Computer.

Program:

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

#define approx_val  2.19328059
#define N 32                /* Number of intervals in each processor */

double integrate_f(double);         /* Integral function */
double simpson(int, double, double, double);

int main(int argc, char *argv[]) {
  int Procs;                /* Number of processors */
  int my_rank;              /* Processor number */
  double total;
  double exact_val_of_Pi, pi, y, processor_output_share[8], x1, x2, l, sum;
  int i;
  MPI_Status status;

  /* Let the system do what it needs to start up MPI */
  MPI_Init(&argc, &argv);

  /* Get my process rank */
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  /* Find out how many processes are being used. */
  MPI_Comm_size(MPI_COMM_WORLD, &Procs);

  /* Each processor computes its interval */
  x1 = ((double) my_rank)/((double) Procs);
  x2 = ((double) (my_rank + 1))/((double) Procs);

  /* l is the same for all processes. */
  l = 1.0/((double) (2 * N * Procs));
  sum = 0.0;
  for(i = 1; i < N ; i++)
  {
    y = x1 + (x2 - x1)*((double) i)/((double) N);

    /* call Simpson's rule  */
    sum = (double) simpson(i, y, l, sum);
  }

  /* Include the endpoints of the intervals */
```

```
  sum += (integrate_f(x1) + integrate_f(x2))/2.0;
  total = sum;

  /* Add up the integrals calculated by each process. */
  if(my_rank == 0)
  {
    processor_output_share[0] = total;

    /* source = i, tag = 0 */
    for(i = 1; i < Procs; i++)
      MPI_Recv(&(processor_output_share[i]), 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &status);
  }
  else
  {
    /* dest = 0, tag = 0 */
    MPI_Send(&total, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
  }

  /* Add up the value of Pi and print the result.  */
  if(my_rank == 0)
  {
    for(i = 0; i < Procs; i++)
      pi += processor_output_share[i];
    pi *= 2.0 * l/ 3.0;
    printf("-----------------------------------------------------\n");
    printf("The computed Pi of the integral for %d grid points is  %25.16e \n",
          (N * Procs), pi);

 /* This is directly derived from the integeration of the formula. See
  the report. */
#if 1
      exact_val_of_Pi = 4.0 * atan(1.0);
#endif

#if 0
      exact_val_of_Pi = 4.0 * log(approx_val);
#endif
      printf("The error or the discrepancy between exact and computed value of Pi : %25.16e\n",
            fabs(pi - exact_val_of_Pi));
      printf("-----------------------------------------------------\n");
  }

  MPI_Finalize();
}


double integrate_f(double x) {
  /* compute and return value */
  return 4.0/(1.0 + x * x);
}

double simpson(int i, double y, double l, double sum) {
  /* store result in sum */
  sum += integrate_f(y);
  sum += 2.0 * integrate_f(y - l);
  if(i == (N - 1))
    sum += 2.0 * integrate_f(y + l);
```

```
  return sum;
} /* simpson */

/*
```

Exercise:-

1. Write a parallel program to compute the value of pi using Simpson's

EXPERIMENT NO: 5

Aim: - Write a program to compute f (x) using the Lagrange's interpolation defined as follows :

$$f(x) = L_0 y_0 + L_1 y_1 + L_2 y_2 + \cdots + L_{n-1} y_{n-1}$$

where

$$L_i = \frac{(x - x_0)(x - x_1)(x - x_2)\cdots(x - x_{i-1})(x - x_{i+1})\cdots(x - x_{n-1})}{(x_i - x_0)(x_i - x_1)(x_i - x_2)\cdots(x_i - x_{i-1})(x_i - x_{i+1})\cdots(x_i - x_{n-1})}$$

Software Required: - MPI Library, GCC Compiler, Linux operating system, Personal Computer.

Program:

```
#include<stdio.h>
#include<string.h>
#include"mpi.h"
void main(int argc,char*argv[])
{
    int my_rank,p,source,dest,tag1=1,tag2=2,tag3=3,tag4=4,tag5=5,tag6=6,tag7=7;
    //char message[100];
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    float a[100];
    float b[100];
    int n;
    if(my_rank==0)
    {
        printf("enter number of elemenets: \n");
        scanf("%d",&n);
        int i;
    //  printf("enter elements: ");
        for(i=0;i<n;i++)
        {
            a[i]=i+1;
            b[i]=(n/2)-i;
            printf("%f,%f\n",a[i],b[i]);
        }
        float x;
        printf("enter value of x:\n");
        scanf("%f",&x);
        int c=n/p;
        int rem=n-(c*(p-1));
        float msum=0.0;
        int j;
        for(i=0;i<rem;i++)
        {
            float z=1.0;
            for(j=0;j<n;j++)
            {
                if(j!=i)
                {
```

```
                        z=(float)z*((x-a[j])/(a[i]-a[j]));
                    }
                }
                msum=msum+z*b[i];
            }
            for(dest=1;dest<p;dest++)
            {
                //int b[20];
                i=rem;
                //for(i=rem;i<rem+c;i++)
                //b[k++]=a[i];
                rem=rem+c;
                MPI_Send(&i,1,MPI_INT,dest,tag1,MPI_COMM_WORLD);
                MPI_Send(&c,1,MPI_INT,dest,tag2,MPI_COMM_WORLD);
                MPI_Send(&x,1,MPI_FLOAT,dest,tag5,MPI_COMM_WORLD);
                MPI_Send(a,100,MPI_FLOAT,dest,tag4,MPI_COMM_WORLD);
                MPI_Send(b,100,MPI_FLOAT,dest,tag6,MPI_COMM_WORLD);
                MPI_Send(&n,1,MPI_INT,dest,tag7,MPI_COMM_WORLD);
                float y;
                MPI_Recv(&y,1,MPI_FLOAT,dest,tag3,MPI_COMM_WORLD,&status);
                msum+=y;
            }
            printf("function value is %f\n",msum);
        }
        else
        {
            source=0;
            //int rec[100];
            int i,c,n;
            float x;
            MPI_Recv(&i,1,MPI_INT,source,tag1,MPI_COMM_WORLD,&status);
            MPI_Recv(&c,1,MPI_INT,source,tag2,MPI_COMM_WORLD,&status);
            MPI_Recv(&x,1,MPI_FLOAT,source,tag5,MPI_COMM_WORLD,&status);
            MPI_Recv(a,100,MPI_FLOAT,source,tag4,MPI_COMM_WORLD,&status);
            MPI_Recv(b,100,MPI_FLOAT,source,tag6,MPI_COMM_WORLD,&status);
            MPI_Recv(&n,1,MPI_INT,source,tag7,MPI_COMM_WORLD,&status);
            int j,k;
            float ssum=0.0;
            for(k=i;k<i+c;k++)
            {
                float z=1.0;
                for(j=0;j<n;j++)
                {
                    if(k!=j)
                    {
                        z=(float)z*((x-a[j])/(a[k]-a[j]));
                    }
                }
                ssum+=z*b[k];
            }
            MPI_Send(&ssum,1,MPI_FLOAT,source,tag3,MPI_COMM_WORLD);

        }
        MPI_Finalize();
}
```

EXPERIMENT NO: 6

Aim: - Write a program for the prefix computation of given data set d0, d1, …, dn-1 on a 2-D mesh network, such that

$$P_i = d_0 + d_1 + d_2 + \ldots + d_i, \quad 0 \leq i < n$$

Software Required: - MPI Library, GCC Compiler, Linux operating system, Personal Computer.

Program:

```
#include<stdio.h>
#include<string.h>
#include<math.h>
#include"mpi.h"
void main(int argc,char*argv[])
{
    int my_rank,p,source,dest,tag1=1,tag2=2,tag3=3,tag4=4,tag5=5,tag6=6,tag7=7,tag8=8;
    //char message[100];
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    int a[100];
    int n;
    if(my_rank==0)
    {
        printf("enter number of elemenets: \n");
        scanf("%d",&n);
        int i;
    //  printf("enter elements: ");
        for(i=0;i<n;i++)
        a[i]=i+1;
        int j=2;
        for(i=0;j<=n;)
        {
            int f;
            for(dest=1;dest<p;dest++)
            {
                int y=1<<i;
                int src=dest-y;
                MPI_Send(&src,1,MPI_INT,dest,tag6,MPI_COMM_WORLD);

            }
            int x=1<<i;
            dest=0+x;
            MPI_Send(&i,1,MPI_INT,dest,tag1,MPI_COMM_WORLD);
            MPI_Send(a,100,MPI_INT,dest,tag4,MPI_COMM_WORLD);
            MPI_Recv(a,100,MPI_INT,dest,tag3,MPI_COMM_WORLD,&status);
            //for(dest=1;dest<p;dest++)
            //MPI_Recv(&f,1,dest,tag8,MPI_COMM_WORLD,&status);
            i++;
            j=1<<(i+1);
        }
```

```
            printf("prefix sum in process %d is %d\n",my_rank,a[0]);
        }
        else
        {
            source=0;
            int src;
            MPI_Recv(&src,1,MPI_INT,source,tag6,MPI_COMM_WORLD,&status);
            if(src>=0)
            if(src>=0)
            {
            source=src;
            int i;
            MPI_Recv(&i,1,MPI_INT,source,tag1,MPI_COMM_WORLD,&status);
            MPI_Recv(a,100,MPI_INT,source,tag4,MPI_COMM_WORLD,&status);
            int x=1<<i;
            dest=my_rank+x;
            if(dest<p)
            {
                MPI_Send(&i,1,MPI_INT,dest,tag2,MPI_COMM_WORLD);
                MPI_Send(a,100,MPI_INT,dest,tag5,MPI_COMM_WORLD);
                MPI_Recv(a,100,MPI_INT,dest,tag7,MPI_COMM_WORLD,&status);
            }
            a[my_rank]+=a[source];
            int j=1<<(i+1);
            if(j==p)
            {
                printf("prefix sum in process %d is %d\n",my_rank,a[my_rank]);
            }
            MPI_Send(a,100,MPI_INT,source,tag3,MPI_COMM_WORLD);
            }
        }
        MPI_Finalize();
}
```

EXPERIMENT NO: 7

Aim: - Write a program to implement the following matrix-matrix multiplication:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}\ ,\ 0 \le i, j < n$$

Software Required: - MPI Library, GCC Compiler, Linux operating system, Personal Computer.

Program:

```
#include <stdio.h>
#include "mpi.h"
#define N  4        /* number of rows and columns in matrix */

MPI_Status status;

double a[N][N],b[N][N],c[N][N];

main(int argc, char **argv)
{
 int numtasks,taskid,numworkers,source,dest,rows,offset,i,j,k;

 struct timeval start, stop;

 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
 MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

 numworkers = numtasks-1;

 /*---------------------------- master ----------------------------*/
 if (taskid == 0) {
  for (i=0; i<N; i++) {
   for (j=0; j<N; j++) {
    a[i][j]= 1.0;
    b[i][j]= 2.0;
   }
  }

  gettimeofday(&start, 0);

  /* send matrix data to the worker tasks */
  rows = N/numworkers;
  offset = 0;

  for (dest=1; dest<=numworkers; dest++)
  {
   MPI_Send(&offset, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
   MPI_Send(&rows, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
   MPI_Send(&a[offset][0], rows*N, MPI_DOUBLE,dest,1, MPI_COMM_WORLD);
   MPI_Send(&b, N*N, MPI_DOUBLE, dest, 1, MPI_COMM_WORLD);
   offset = offset + rows;
  }
```

```
  /* wait for results from all worker tasks */
  for (i=1; i<=numworkers; i++)
  {
    source = i;
    MPI_Recv(&offset, 1, MPI_INT, source, 2, MPI_COMM_WORLD, &status);
    MPI_Recv(&rows, 1, MPI_INT, source, 2, MPI_COMM_WORLD, &status);
    MPI_Recv(&c[offset][0], rows*N, MPI_DOUBLE, source, 2, MPI_COMM_WORLD, &status);
  }

  gettimeofday(&stop, 0);

  printf("Here is the result matrix:\n");
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++)
      printf("%6.2f   ", c[i][j]);
    printf ("\n");
  }

  fprintf(stdout,"Time = %.6f\n\n",
      (stop.tv_sec+stop.tv_usec*1e-6)-(start.tv_sec+start.tv_usec*1e-6));

 }

 /*---------------------------- worker----------------------------*/
 if (taskid > 0) {
  source = 0;
  MPI_Recv(&offset, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
  MPI_Recv(&rows, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
  MPI_Recv(&a, rows*N, MPI_DOUBLE, source, 1, MPI_COMM_WORLD, &status);
  MPI_Recv(&b, N*N, MPI_DOUBLE, source, 1, MPI_COMM_WORLD, &status);

  /* Matrix multiplication */
  for (k=0; k<N; k++)
    for (i=0; i<rows; i++) {
      c[i][k] = 0.0;
      for (j=0; j<N; j++)
        c[i][k] = c[i][k] + a[i][j] * b[j][k];
    }


  MPI_Send(&offset, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
  MPI_Send(&rows, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
  MPI_Send(&c, rows*N, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD);
 }

 MPI_Finalize();
}
```

EXPERIMENT NO: 8

Aim: - Write a program to compute the polynomial using shuffle and exchange network.

$$f = a_0 x^0 + a_1 x^1 + a_2 x^2 + \square\ a_{n-1} x^{n-1}$$

Software Required: - MPI Library, GCC Compiler, Linux operating system, Personal Computer.

Program:

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

int routingFn(int j, int i){
	return j+pow(2,i);
}

int routingFn2(int j){
	int t;
	t = j>>2;
	j = (j<<1);
	j = (j%8)|t;
	return j;
}


void main(int argc, char* argv[]){
	int my_rank,p,source,dest,tag=0;
	int tag2=1;
	MPI_Status status;
	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
	MPI_Comm_size(MPI_COMM_WORLD, &p);
	int n;
	int my_value;
	int my_step;
	int total_step = log2(p);
	int x=1;
	if(my_rank==0){
		my_step=0;
		int arr[] = {1,1,1,1,1,1,1,1};
		int mask[] = {0,1,0,1,0,1,0,1};
		int mask_n[] = {0,0,0,0,0,0,0,0};
		my_value=arr[0];
		int i;
		for(my_step=0;my_step<total_step;my_step++){
			for(i=1;i<p;i++){
				int t=arr[i];
				int m=mask[i];
				MPI_Send(&t, 1, MPI_INT,i,tag,MPI_COMM_WORLD);
				MPI_Send(&m, 1, MPI_INT,i,tag+1,MPI_COMM_WORLD);
			}
```

```
            if(mask[my_rank]){
                arr[0] *= x;
            }
            for(i=1;i<p;i++){
                int y;
                MPI_Recv(&y,1,MPI_INT,i,tag+2,MPI_COMM_WORLD,&status);
                arr[i] = y;
            }

            for(i=0;i<p;i++){
                int t=routingFn2(i);
                mask_n[t]=mask[i];
            }
            for(i=0;i<p;i++){
                mask[i]=mask_n[i];
                printf("%d ",mask[i]);
            }
            printf("\n");
        }
        my_value = arr[0];

        for(my_step=0;my_step<total_step;my_step++){
            for(i=1;i<p;i++){
                int t = arr[i];
                MPI_Send(&t, 1, MPI_INT,i,tag,MPI_COMM_WORLD);
                MPI_Send(&my_step,1,MPI_INT,i,tag+1,MPI_COMM_WORLD);
            }

            int ns;
            ns = routingFn(my_rank,my_step);
            int k = arr[my_rank];
            MPI_Send(&k, 1, MPI_INT,ns,tag+3,MPI_COMM_WORLD);
            for(i=1;i<p;i++){
                int y;
                MPI_Recv(&y,1,MPI_INT,i,tag+2,MPI_COMM_WORLD,&status);

                arr[i] = y;
            }
        }
        for(i=0;i<p-1;i++){
            printf("Result for P[%d] : %d\n",i,arr[i]);
        }
    }
    else{
        int j;
        for(j=0;j<total_step;j++){
            source = 0;
            MPI_Recv(&my_value,1,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
            MPI_Recv(&my_step,1,MPI_INT,source,tag+1,MPI_COMM_WORLD,&status);
            if(my_step){
                my_value*=x;
            }
            MPI_Send(&my_value,1,MPI_INT,source,tag+2,MPI_COMM_WORLD);
            x*=x;
        }

        for(j=0;j<total_step;j++){
```

```
            source = 0;
            MPI_Recv(&my_value,1,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
            MPI_Recv(&my_step,1,MPI_INT,source,tag+1,MPI_COMM_WORLD,&status);
            int ns;
            if(my_rank%2==0){
                ns = routingFn(my_rank,my_step);
                if(ns<8){
                    MPI_Send(&my_value, 1, MPI_INT,ns,tag+3,MPI_COMM_WORLD);
                }
                int incoming;
                int incomingP = my_rank - pow(2,my_step);
                if(my_rank>=pow(2,my_step)){
                    MPI_Recv(&incoming, 1, MPI_INT,incomingP,tag+3,MPI_COMM_WORLD,&status);
                }

                my_value += incoming;

                MPI_Send(&my_value, 1, MPI_INT,source,tag+2,MPI_COMM_WORLD);
            }
            else{
                ns = routingFn(my_rank,my_step);
                int incoming;
                int incomingP = my_rank - pow(2,my_step);
                if(my_rank>=pow(2,my_step)){
                    MPI_Recv(&incoming, 1, MPI_INT,incomingP,tag+3,MPI_COMM_WORLD,&status);
                }
                if(ns<8){
                    MPI_Send(&my_value, 1, MPI_INT,ns,tag+3,MPI_COMM_WORLD);
                }
                my_value += incoming;

                MPI_Send(&my_value, 1, MPI_INT,source,tag+2,MPI_COMM_WORLD);
            }
        }

    }

    MPI_Finalize();

}
```

EXPERIMENT NO: 9

Aim: - Write a program to implement the following matrix-vector multiplication:

$$c_k = a_{k0}x_0 + a_{k1}x_1 + \square + a_{k,n-1}x_{n-1}, \quad k = 0, 1, \ldots, n\text{-}1$$

Software Required: - MPI Library, GCC Compiler, Linux operating system, Personal Computer.

Program:

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

void multi(int Count,float *Sum,float Vec[],float Data[],int Column)
{
  int i=0,j=0,k=0;
  while(i<Count)
  {
    Sum[i]=0;
    for(j=0;j<Column;j++)
    {
      Sum[i] = Sum[i] + Data[k] * Vec[j];
      k++;
    }
   i++;
  }
}

int main(int argc,char *argv[])
{
  int rank,size,*sendcount,*displace,*reccount;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Status status;
  FILE *fp;
  char c;
  int i,j,k=0,count=0,row=0,column=0;
  float n=0,*sum,*rec_data,*data,*vec;
  sendcount = (int*)calloc(sizeof(int),size);
  reccount  = (int*)calloc(sizeof(int),size);
  displace  = (int*)calloc(sizeof(int),size);
  if(rank==0)
  {
    fp=fopen("matrix.txt","r");
    while(fscanf(fp,"%f",&n)!=-1)
    {
      c=fgetc(fp);
      if(c=='\n'){ row=row+1; }
      count++;
    }
    column=count/row;
    printf("Row=%d column=%d proc=%d\n",row,column,size);
    float mat[row][column];
```

```
 fseek( fp, 0, SEEK_SET );
 data = (float*)calloc(sizeof(float),row*column);
 vec  = (float*)calloc(sizeof(float),column);
 for(i=0;i<row;i++)
 {
   for(j=0;j<column;j++)
   {
     fscanf(fp,"%f",&mat[i][j]);
     data[k] = mat[i][j];
     k++;
   }
 }
 fclose(fp);
 fp    =  fopen("vector.txt","r");
 count = 0;
 while(fscanf(fp,"%f",&n)!=-1){  count++; }
 printf("length of vector = %d\n",count);
 if(column!=count) { printf("Dimensions do not match.\nCode Terminated"); MPI_Abort(MPI_COMM_WORLD,0); }
 fseek( fp, 0, SEEK_SET );
 for(i=0;i<column;i++)
 {
  fscanf(fp,"%f",&vec[i]);
 }
 fclose(fp);
 count=0;
 while(1)
 {
  for(i=0;i<size;i++)
  {
    sendcount[i] = sendcount[i]+1;
    count++;
    if(count==row) break;
  }
  if(count==row)  break;
 }
 for(i=1;i<size;i++)
 {
  displace[i] = displace[i-1] + sendcount[i-1]*column;
  sendcount[i-1] = sendcount[i-1] * column;
 }
 sendcount[size-1] = sendcount[size-1] * column;
 for(i=0;i<size;i++)
   printf("sendcout=%d disp=%d\n",sendcount[i],displace[i]);
}
MPI_Bcast(&row,1,MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast(&column,1,MPI_INT,0,MPI_COMM_WORLD);
if(rank!=0)
{
 //data=(float*)calloc(sizeof(float),row*column);
 vec = (float *)malloc(sizeof(float) * column);
}
MPI_Bcast(vec,column,MPI_FLOAT,0,MPI_COMM_WORLD);
MPI_Bcast(sendcount,size,MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast(displace,size,MPI_INT,0,MPI_COMM_WORLD);
rec_data=(float*)calloc(sizeof(float),sendcount[rank]);
//MPI_Bcast(data,row*column,MPI_INT,0,MPI_COMM_WORLD);
MPI_Scatterv(data,sendcount,displace,MPI_FLOAT,rec_data,sendcount[rank],MPI_FLOAT,0,MPI_COMM_WORLD);
```

```
  count=sendcount[rank]/column;
  sum=(float*)calloc(sizeof(float),count);
  multi(count,sum,vec,rec_data,column);
  float *result=(float *)calloc(sizeof(float),row);
  int disp[size];
  disp[0]=0;
  reccount[0]=sendcount[0]/column;
  for(i=1;i<size;i++)
  {
    disp[i] = disp[i-1] + sendcount[i-1]/column;
    reccount[i]=sendcount[i]/column;
  }
  MPI_Gatherv(sum,count,MPI_FLOAT,result,reccount,disp,MPI_FLOAT,0,MPI_COMM_WORLD);
  if(rank==0)
  {
    printf("\nMatrix Vector Multiplication is:\n");
    for(i=0;i<row;i++)
    {
       printf("%.3f\n",result[i]);
    }
  }
  free(vec);
  free(sum);
  free(sendcount);
  free(displace);
  free(reccount);
  free(rec_data);
  MPI_Finalize();
  return 0;
}
```

Question:

1. What run-times does your system get for matrix-vector multiplication?

EXPERIMENT NO: 10

Aim: - Write a program to calculate the definite integral of a nonnegative function f (x) using trapezoidal rule with broadcast and reduce functions for message sharing among PEs.

Software Required: - MPI Library, GCC Compiler, Linux operating system, Personal Computer.

Theory: The trapezoidal rule is a technique for approximating the region under a function, , using trapezoids to calculate area. The process is quite simple. Let a and b represent the left and right endpoints of the function. The interval [a,b] is divided into subintervals. For each subinterval, the function is approximated with a straight line between the function values at both ends of the subinterval. Each subinterval is now a trapezoid. Lastly, the area of each trapezoid is calculated and all areas are summed to get an approximation of the area under the function.

Program:

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

const double a = 0;
const double b = 2000000000;

/* Function declarations */
void Get_input(int argc, char* argv[], int my_rank, double* n_p);
double Trap(double left_endpt, double right_endpt, int trap_count,
   double base_len);
double f(double x);

int main(int argc, char** argv) {
   int my_rank, comm_sz, local_n;
   double n, h, local_a, local_b;
   double local_int, total_int;
   double start, finish, loc_elapsed, elapsed;

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

   Get_input(argc, argv, my_rank, &n); /*Read user input */

   /*Note: h and local_n are the same for all processes*/
   h = (b-a)/n;          /* length of each trapezoid */
   local_n = n/comm_sz;  /* number of trapezoids per process */

   /* Length of each process' interval of integration = local_n*h. */
   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;

   MPI_Barrier(MPI_COMM_WORLD);
```

```
  start = MPI_Wtime();
  /* Calculate each process' local integral using local endpoints*/
  local_int = Trap(local_a, local_b, local_n, h);
  finish = MPI_Wtime();
  loc_elapsed = finish-start;
  MPI_Reduce(&loc_elapsed, &elapsed, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

  /* Add up the integrals calculated by each process */
  MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);


  if (my_rank == 0) {
    printf("With n = %.0f trapezoids, our estimate\n", n);
    printf("of the integral from %.0f to %.0f = %.0f\n",
       a, b, total_int);
    printf("Elapsed time = %f milliseconds \n", elapsed * 1000);
  }

  /* Shut down MPI */
  MPI_Finalize();

  return 0;
} /*  main  */


void Get_input(int argc, char* argv[], int my_rank, double* n_p){
    if (my_rank == 0) {
            if (argc!= 2){
               fprintf(stderr, "usage: mpirun -np <N> %s <number of trapezoids> \n", argv[0]);
        fflush(stderr);
        *n_p = -1;
            } else {
                *n_p = atoi(argv[1]);
            }
    }
    // Broadcasts value of n to each process
    MPI_Bcast(n_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    // negative n ends the program
   if (*n_p <= 0) {
      MPI_Finalize();
      exit(-1);
   }
}  /* Get_input */

*/
double Trap(double left_endpt, double right_endpt, int trap_count, double base_len) {
  double estimate, x;
  int i;

  estimate = (f(left_endpt) + f(right_endpt))/2.0;
  for (i = 1; i <= trap_count-1; i++) {
    x = left_endpt + i*base_len;
    estimate += f(x);
  }
  estimate = estimate*base_len;
```

```
  return estimate;
} /*  Trap  */


double f(double x) {
  return x*x;
} /* f */
```

Discussion:

1. Each process calculates "its" interval of integration.
2. Each process estimates the integral of f(x) over its interval using the trapezoidal rule.

3a. Each process != 0 sends its integral to 0.

3b. Process 0 sums the calculations received from the individual processes and prints the result.

EXPERIMENT NO: 11

Aim: - Write a program to implement matrix-matrix multiplication on a 2D wrap around mesh network.

Software Required: - MPI Library, GCC Compiler, Linux operating system, Personal Computer.

Program:

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

void main(int argc, char* argv[] )
{
    int my_rank,p,a[100][100],b[100][100],source,dest;
    int tag=0;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    if(my_rank!=0)
    {
        int x,y,z,x1,y1,z1;
        MPI_Recv(&x,1,MPI_INT,0,tag, MPI_COMM_WORLD, &status);
        MPI_Recv(&y,1,MPI_INT,0,tag, MPI_COMM_WORLD, &status);
        MPI_Recv(&z,1,MPI_INT,0,tag, MPI_COMM_WORLD, &status);
        int n=sqrt(p-1);
        int k;
        int i=(my_rank/n)+1;
        if(my_rank%n==0)
        i--;
        int j=(my_rank%n);
        if(my_rank%n==0)
        j=n;
        for(k=1;k<=n-1;k++)
        {

            if(i>k)
            {
                int dest=my_rank-1;
                if(dest%n==0)
                dest=dest+n;
                int source=my_rank+1;
                if(source%n==1)
                source=source-n;
                MPI_Send(&x,1,MPI_INT,dest,tag,MPI_COMM_WORLD);
                MPI_Recv(&x,1,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
            }
            if(j>k)
            {
                int dest=my_rank-n;
                if(dest<=0)
                dest=dest+n*n;
```

```
                int source=my_rank+n;
                if(source>n*n)
                source=source-n*n;
                MPI_Send(&y,1,MPI_INT,dest,tag,MPI_COMM_WORLD);
                MPI_Recv(&y,1,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
            }
        }
        for(k=1;k<=n;k++)
        {
            z=z+x*y;
            int dest=my_rank-1;
            if(dest%n==0)
            dest=dest+n;
            int source=my_rank+1;
            if(source%n==1)
            source=source-n;
            MPI_Send(&x,1,MPI_INT,dest,tag,MPI_COMM_WORLD);
            MPI_Recv(&x,1,MPI_INT,source,tag,MPI_COMM_WORLD,&status);

            dest=my_rank-n;
            if(dest<=0)
            dest=dest+n*n;
            source=my_rank+n;
            if(source>n*n)
            source=source-n*n;
            MPI_Send(&y,1,MPI_INT,dest,tag,MPI_COMM_WORLD);
            MPI_Recv(&y,1,MPI_INT,source,tag,MPI_COMM_WORLD,&status);

        }
        MPI_Send(&z,1,MPI_INT,0,tag,MPI_COMM_WORLD);
}
else
{
        int n;
        printf("Enter dimension of matrix: \n");
        scanf("%d",&n);
        int i,j;
        int count=0;
        for(i=0;i<n;i++)
        {
            for(j=0;j<n;j++)
            {
                count++;
                if(j==i)
                a[i][j]=1;
                else
                a[i][j]=0;

                b[i][j]=2;
                int x=a[i][j];
                int y=b[i][j];
                int z=0;
                MPI_Send(&x,1,MPI_INT,count,tag,MPI_COMM_WORLD);
                MPI_Send(&y,1,MPI_INT,count,tag,MPI_COMM_WORLD);
                MPI_Send(&z,1,MPI_INT,count,tag,MPI_COMM_WORLD);
            }
        }
```

```
        int c[100][100];
        for(i=1;i<=n;i++)
        {
            for(j=1;j<=n;j++)
            {
                int source=(i-1)*n+j;
                int x;
                MPI_Recv(&x,1,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
                c[i-1][j-1]=x;
                printf("%d ",x);
            }
            printf("\n");
        }
    }
    MPI_Finalize();
}
```

EXPERIMENT NO: 12

Aim: Write a program to implement odd-even transposition sort on a linear array.

Software required: -MPI Library, GCC Compiler, Linux operating system, Personal Computer.

Program:

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

void main(int argc, char* argv[] )
{
    int my_rank,p,a[100],source,dest;
    int tag=0;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    if(my_rank!=0)
    {
        int x,z;
        MPI_Recv(&x,1,MPI_INT,0,tag, MPI_COMM_WORLD, &status);
        int j;
        int k=(p-1)/2;
        if((p-1)%2!=0)
        k++;
        for(j=1;j<=k;j++)
        {
            if(my_rank+1<p && my_rank%2==1)
            {
                MPI_Send(&x,1,MPI_INT,my_rank+1,tag,MPI_COMM_WORLD);
                MPI_Recv(&x,1,MPI_INT,my_rank+1,tag,MPI_COMM_WORLD,&status);
            }
            if(my_rank-1>=1 && my_rank%2==0)
            {
                MPI_Recv(&z,1,MPI_INT,my_rank-1,tag,MPI_COMM_WORLD,&status);
                if(z>x)
                {
                    int temp=z;
                    z=x;
                    x=temp;
                }
                MPI_Send(&z,1,MPI_INT,my_rank-1,tag,MPI_COMM_WORLD);
            }
            if(my_rank+1<p && my_rank%2==0)
            {
                MPI_Send(&x,1,MPI_INT,my_rank+1,tag,MPI_COMM_WORLD);
                MPI_Recv(&x,1,MPI_INT,my_rank+1,tag,MPI_COMM_WORLD,&status);
            }
            if(my_rank-1>=1 && my_rank%2==1)
            {
                MPI_Recv(&z,1,MPI_INT,my_rank-1,tag,MPI_COMM_WORLD,&status);
                if(z>x)
```

```
                {
                    int temp=z;
                    z=x;
                    x=temp;

                }
                MPI_Send(&z,1,MPI_INT,my_rank-1,tag,MPI_COMM_WORLD);
            }
        }
        MPI_Send(&x,1,MPI_INT,0,tag,MPI_COMM_WORLD);
    }
    else
    {
        int n;
        printf("Enter total elements: \n");
        scanf("%d",&n);
        int i;
        for(i=0;i<n;i++)
            scanf("%d",&a[i]);
        for(i=0;i<n;i++)
        {
            int b=a[i];
            MPI_Send(&b,1,MPI_INT,i+1,tag,MPI_COMM_WORLD);
        }
        for(source=1;source<p;source++)
        {
            int x;
            MPI_Recv(&x,1,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
            a[source-1]=x;
            printf("Message from process %d element = %d\n",source-1,x);
        }
    }
    MPI_Finalize();
}
```

CUDA PROGRAMMING

Experiment:1

Aim: - Create, compile and run an a CUDA "Hello world" program using remote access over server.
Software Required: NVCC Library, GCC Compiler, Linux operating system, Personal Computer.

CUDA -First Programs: "Hello, world" is traditionally the first program we write. We can do the same for CUDA. Here it is:

In file hello.cu:

```
#include "stdio.h"
int main()
{
 printf("Hello, world\n");
 return 0;
}
```

On your host machine, you can compile and this with:

$ nvcc hello.cu

Execution on GPU equipped server
$ ./a.out

Discussion:
You can change the output file name with the –o flag: nvcc –o hello hello.cu
If you edit your .bashrc file you can also add your current directory to your path if you don't want to have to type the preceding . all of the time, which refers to the current working directory.
Add export PATH=$PATH:.
To the .bashrc file. Some would recommend not doing this for security purposes.

The point is that CUDA C programs can do everything a regular C program can do.

Flow of Program: Open text editor (like vi/vim) open a new file - call it whatever you'd like. It should do the following:

- Use the appropriate .cu include file
- compile and this with: **$ nvcc hello.cu**
- Print a hello message that includes its task rank and processor name **Execution on**
- **GPU equipped server$ ./a.out**
- Terminate the Connection environment

Experiment:2 :

Aim: - Create, compile and run an a CUDA " two numbers together using a kernel function **" program using remote access over server.**

Software Required: NVCC Library, GCC Compiler, Linux operating system, Personal Computer.

```
#include "stdio.h"
__global__ void add(int a, int b, int *c)
{
 *c = a + b;
}
int main()
{
int a,b,c;
int *dev_c;
a=3;
b=4;
cudaMalloc((void**)&dev_c, sizeof(int));
add<<<1,1>>>(a,b,dev_c);
cudaMemcpy(&c, dev_c, sizeof(int), cudaMemcpyDeviceToHost);
printf("%d + %d is %d\n", a, b, c);
cudaFree(dev_c);
return 0;
}
```

Discussion:

cudaMalloc returns cudaSuccess if it was successful; could check to ensure that the program will run correctly.

Example: Summing Vectors

This is a simple problem. Given two vectors (i.e. arrays), we would like to add them together in a third array. For example:
A = {0, 2, 4, 6, 8}
B = {1, 1, 2, 2, 1}
Then A + B =
C = {1, 3, 6, 8, 9}
In this example the array is 5 elements long, so our approach will be to create 5 different threads. The first thread is responsible for computing C[0] = A[0] + B[0]. The second thread is responsible for computing C[1] = A[1] + B[1], and so forth.

Experiment3 :

Aim: - Create, compile and run an a CUDA " two numbers together using a modified kernel function **" program using remote access over server.**

Software Required: NVCC Library, GCC Compiler, Linux operating system, Personal Computer.

Program:

```
#include "stdio.h"
#define N 10
__global__ void add(int *a, int *b, int *c)
{
 int tID = blockIdx.x;
 if (tID < N)
 {
 c[tID] = a[tID] + b[tID];
 }
}
int main()
{
 int a[N], b[N], c[N];
 int *dev_a, *dev_b, *dev_c;
 cudaMalloc((void **) &dev_a, N*sizeof(int));
 cudaMalloc((void **) &dev_b, N*sizeof(int));
 cudaMalloc((void **) &dev_c, N*sizeof(int));
 // Fill Arrays
 for (int i = 0; i < N; i++)
 {
 a[i] = i,
 b[i] = 1;
 }
 cudaMemcpy(dev_a, a, N*sizeof(int), cudaMemcpyHostToDevice);
 cudaMemcpy(dev_b, b, N*sizeof(int), cudaMemcpyHostToDevice);
 add<<<N,1>>>(dev_a, dev_b, dev_c);
 cudaMemcpy(c, dev_c, N*sizeof(int), cudaMemcpyDeviceToHost);
 for (int i = 0; i < N; i++)
 {
 printf("%d + %d = %d\n", a[i], b[i], c[i]);
 }
 return 0;
}
```

Discussion:

blockIDx.x gives us the Block ID, which ranges from 0 to N-1. What if we used add<<<1,N>>> instead? Then we can access by the ThreadID which is the variable threadIDx.x.

Example: let's add two 2D arrays. We can define a 2D array of ints as follows:
int c[2][3];

The following code illustrates how the 2D array is laid out in memory:

```
for (int i=0; i < 2; i++)
        for (int j=0; j< 3; j++)
                printf("[%d][%d] at %ld\n",i,j,&c[i][j]);
Output:
[0][0] at 140733933298160
[0][1] at 140733933298164
[0][2] at 140733933298168
[1][0] at 140733933298172
[1][1] at 140733933298176
[1][2] at 140733933298180
```

We can see that we have a layout where the next cell in the j dimension occupies the next sequential integer in memory, where an int is 4 bytes:

c[0][0] at &c c[0][1] at &c + 4 c[0][2] at &c + 8
c[1][0] at &c + 12 c[1][1] at &c + 16 c[1][2] at &c + 20

In general, the address of a cell can be computed by:

&c + [(sizeof(int) * sizeof-j-dimension * i] + (sizeof(int)) * j

In our example the size of the j dimension is 3.

For example, the cell at c[1][1] would be combined as the base address + (4*3*1) + (4*1) = &c+16.
C will do the addressing for us if we use the array notation, so if INDEX=i*WIDTH + J
then we can access the element via: c[INDEX]
CUDA requires we allocate memory as a one-dimensional array, so we can use the mapping above to a 2D array.

Experiment4 :

Aim: - Create, compile and run an a CUDA " two numbers together with grid constraint **" program using remote access over server.**

Software Required: NVCC Library, GCC Compiler, Linux operating system, Personal Computer.

To make the mapping a little easier in the kernel function we can declare the blocks to be in a grid that is the same dimensions as the 2D array. This will create variables blockIdx.x and blockIdx.y that correspond to the width and height of the array.

```
#include "stdio.h"
#define COLUMNS 3
#define ROWS 2
__global__ void add(int *a, int *b, int *c)
{
 int x = blockIdx.x;
 int y = blockIdx.y;
 int i = (COLUMNS*y) + x;
 c[i] = a[i] + b[i];
}

/* ------------- COMPUTATION DONE ON GPU ----------------------------*/

int main()
{
 int a[ROWS][COLUMNS], b[ROWS][COLUMNS], c[ROWS][COLUMNS];
 int *dev_a, *dev_b, *dev_c;
 cudaMalloc((void **) &dev_a, ROWS*COLUMNS*sizeof(int));
 cudaMalloc((void **) &dev_b, ROWS*COLUMNS*sizeof(int));
 cudaMalloc((void **) &dev_c, ROWS*COLUMNS*sizeof(int));
 for (int y = 0; y < ROWS; y++) // Fill Arrays
 for (int x = 0; x < COLUMNS; x++)
 {
 a[y][x] = x;
 b[y][x] = y;
 }
 cudaMemcpy(dev_a, a, ROWS*COLUMNS*sizeof(int),
cudaMemcpyHostToDevice);
 cudaMemcpy(dev_b, b, ROWS*COLUMNS*sizeof(int),
cudaMemcpyHostToDevice);
 dim3 grid(COLUMNS,ROWS);
 add<<<grid,1>>>(dev_a, dev_b, dev_c);
 cudaMemcpy(c, dev_c, ROWS*COLUMNS*sizeof(int),
cudaMemcpyDeviceToHost);

/* ------------- COMPUTATION DONE ON HOST CPU ----------------------------*/

 for (int y = 0; y < ROWS; y++) // Output Arrays
 {
 for (int x = 0; x < COLUMNS; x++)
 {
 printf("[%d][%d]=%d ",y,x,c[y][x]);
```

```
 }
 printf("\n");
 }
 return 0;
}
```

Flow of Program 2,3,4: Open text editor (like vi/vim) open a new file - call it whatever you'd like. It should do the following:

- Use the appropriate .cu include file
- compile and this with: **$ nvcc addition.cu**
- Print a general value of addition that includes its task rank and processor name **Execution on GPU equipped server$ ./a.out**
- Start modifying to optimize the computation of addition and execute task rank and processor name **Execution on GPU equipped server$ ./a.out**
- Start constraint on grid to optimize the computation of addition and execute task rank and processor name **Execution on GPU equipped server$ ./a.out**
- Terminate the Connection environment

Experiment:5

Aim: - Create, compile and run an a CUDA " Matrix addition **" program using remote access over server.**

.

Software Required: NVCC Library, GCC Compiler, Linux operating system, Personal Computer.

Program:

```
// Matrix addition program MatrixAdd.cu

        #include <stdio.h>
        #include <cuda.h>
        #include <stdlib.h>

        __global__ void gpu_matrixadd(int *a,int *b, int *c, int N) {

           int col = threadIdx.x + blockDim.x * blockIdx.x;
           int row = threadIdx.y + blockDim.y * blockIdx.y;

           int index = row * N + col;

                        if(col < N && row < N)
                c[index] = a[index]+b[index];

        }

        void cpu_matrixadd(int *a,int *b, int *c, int N) {

           int index;
           for(int col=0;col < N; col++)
                        for(int row=0;row < N; row++) {
                                        index = row * N + col;
                                        c[index] = a[index]+b[index];
                        }
        }

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

           char key;

           int i, j;                                                    // loop counters

           int Grid_Dim_x=1, Grid_Dim_y=1;                                      //Grid structure values
           int Block_Dim_x=1, Block_Dim_y=1;                    //Block structure values

           int noThreads_x, noThreads_y;                // number of threads available in device, each dimension
           int noThreads_block;                                 // number of threads in a block
```

```
   int N = 10;                                                    // size of array in each dimension
   int *a,*b,*c,*d;
   int *dev_a, *dev_b, *dev_c;
   int size;                                                      // number of bytes in arrays

   cudaEvent_t start, stop;                          // using cuda events to measure time
   float elapsed_time_ms;                            // which is applicable for asynchronous code also

/* --------------------ENTER INPUT PARAMETERS AND DATA ----------------------*/

do {  // loop to repeat complete program

   __global__ void input_parameter(sizeof(x), sizeof(y), nub_block))


                x = (int*) malloc(size);                  //this time use dynamically allocated memory for arrays on
host
                y = (int*) malloc(size);
                nub_block = (int*) malloc(size);          // results from GPU
                // results from CPU

   for(i=0;i < N;i++)                                // load arrays with some numbers
   for(j=0;j < N;j++) {
                a[i * N + j] = i;
                b[i * N + j] = i;
   }
```

Experiment:6

Aim: - Create, compile and run an a CUDA " Matrix addition **" program using remote access over server.**

Software Required: NVCC Library, GCC Compiler, Linux operating system, Personal Computer.

Program:

```
/* ------------- COMPUTATION DONE ON GPU ----------------------------*/

   cudaMalloc((void**)&dev_a, size);                        // allocate memory on device
   cudaMalloc((void**)&dev_b, size);
   cudaMalloc((void**)&dev_c, size);

   cudaMemcpy(dev_a, a , size ,cudaMemcpyHostToDevice);
   cudaMemcpy(dev_b, b , size ,cudaMemcpyHostToDevice);
   cudaMemcpy(dev_c, c , size ,cudaMemcpyHostToDevice);

   cudaEventCreate(&start);                     // instrument code to measure start time
   cudaEventCreate(&stop);

   cudaEventRecord(start, 0);
//  cudaEventSynchronize(start);       // Needed?

   gpu_matrixadd<<<Grid,Block>>>(dev_a,dev_b,dev_c,N);

   cudaMemcpy(c,dev_c, size ,cudaMemcpyDeviceToHost);

   cudaEventRecord(stop, 0);          // instrument code to measue end time
   cudaEventSynchronize(stop);
   cudaEventElapsedTime(&elapsed_time_ms, start, stop );

//  for(i=0;i < N;i++)
//  for(j=0;j < N;j++)
//     printf("%d+%d=%d\n",a[i * N + j],b[i * N + j],c[i * N + j]);

   printf("Time to calculate results on GPU: %f ms.\n", elapsed_time_ms);  // print out execution time

/* ------------- COMPUTATION DONE ON HOST CPU ----------------------------*/

   cudaEventRecord(start, 0);                   // use same timing
//  cudaEventSynchronize(start);       // Needed?

   cpu_matrixadd(a,b,d,N);                      // do calculation on host

   cudaEventRecord(stop, 0);          // instrument code to measue end time
   cudaEventSynchronize(stop);
   cudaEventElapsedTime(&elapsed_time_ms, start, stop );

   printf("Time to calculate results on CPU: %f ms.\n", elapsed_time_ms);  // print out execution time
}
```

Flow of Program : Open text editor (like vi/vim) open a new file - call it whatever you'd like. It should do the following:

- Use the appropriate .cu include file
- Enter input parameters and data
- Computation done on GPU
- Computation done on CPU
- Call function in main
- Write connection termination segment
- Compile and this with: **$ nvcc**
- Print a general value of addition and execute task rank and processor name **Execution on GPU equipped server$ ./a.out**

Terminate the Connection environment

Experiment:7

Aim: - Create, compile and run an a CUDA " Matrix multiplication **" program using remote access over server.**

Program:

```
// Matrix addition program MatrixMul.cu
// written by Barry Wilkinson, UNC-Charlotte. December 27, 2010.

#include <stdio.h>
#include <cuda.h>
#include <stdlib.h>

__global__ void gpu_ MatrixMul(int *a,int *b, int *c, int N) {

   int col = threadIdx.x + blockDim.x * blockIdx.x;
   int row = threadIdx.y + blockDim.y * blockIdx.y;

   int index = row * N + col;

                if(col < N && row < N)
        c[index] = a[index]+b[index];

}

void cpu_ MatrixMul (int *a,int *b, int *c, int N) {

   int index;
   for(int col=0;col < N; col++)
                for(int row=0;row < N; row++) {
                        index = row * N + col;
                        c[index] = a[index]+b[index];
                }
}

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

   char key;

   int i, j;                                            // loop counters

   int Grid_Dim_x=1, Grid_Dim_y=1;                              //Grid structure values
   int Block_Dim_x=1, Block_Dim_y=1;                    //Block structure values

   int noThreads_x, noThreads_y;                // number of threads available in device, each dimension
   int noThreads_block;                                 // number of threads in a block

   int N = 10;                                                  // size of array in each dimension
   int *a,*b,*c,*d;
   int *dev_a, *dev_b, *dev_c;
   int size;                                            // number of bytes in arrays
```

```
   cudaEvent_t start, stop;                    // using cuda events to measure time
   float elapsed_time_ms;                      // which is applicable for asynchronous code also

/* --------------------ENTER INPUT PARAMETERS AND DATA -----------------------*/

do {  // loop to repeat complete program

   printf ("Device characteristics -- some limitations (compute capability 1.0)\n");
   printf ("               Maximum number of threads per block = 512\n");
   printf ("               Maximum sizes of x- and y- dimension of thread block = 512\n");
   printf ("               Maximum size of each dimension of grid of thread blocks = 65535\n");

   printf("Enter size of array in one dimension (square array), currently %d\n",N);
   scanf("%d",&N);

   do {
               printf("\nEnter nuumber of blocks per grid in x dimension), currently %d  : ",Grid_Dim_x);
               scanf("%d",&Grid_Dim_x);

               printf("\nEnter nuumber of blocks per grid in y dimension), currently %d  : ",Grid_Dim_y);
               scanf("%d",&Grid_Dim_y);

               printf("\nEnter nuumber of threads per block in x dimension), currently %d  : ",Block_Dim_x);
               scanf("%d",&Block_Dim_x);

               printf("\nEnter nuumber of threads per block in y dimension), currently %d  : ",Block_Dim_y);
               scanf("%d",&Block_Dim_y);

               noThreads_x = Grid_Dim_x * Block_Dim_x;                 // number of threads in x dimension
               noThreads_y = Grid_Dim_y * Block_Dim_y;                 // number of threads in y dimension

               noThreads_block = Block_Dim_x * Block_Dim_y;        // number of threads in a block

               if (noThreads_x < N) printf("Error -- number of threads in x dimension less than number of elements in
arrays, try again\n");
               else if (noThreads_y < N) printf("Error -- number of threads in y dimension less than number of elements in
arrays, try again\n");
               else if (noThreads_block > 512) printf("Error -- too many threads in block, try again\n");
               else printf("Number of threads not used = %d\n", noThreads_x * noThreads_y - N * N);

   } while (noThreads_x < N || noThreads_y < N || noThreads_block > 512);

   dim3 Grid(Grid_Dim_x, Grid_Dim_x);                  //Grid structure
   dim3 Block(Block_Dim_x,Block_Dim_y);    //Block structure, threads/block limited by specific device

   size = N * N * sizeof(int);                 // number of bytes in total in arrays

   a = (int*) malloc(size);                    //this time use dynamically allocated memory for arrays on host
   b = (int*) malloc(size);
   c = (int*) malloc(size);                    // results from GPU
   d = (int*) malloc(size);                    // results from CPU

   for(i=0;i < N;i++)                          // load arrays with some numbers
   for(j=0;j < N;j++) {
               a[i * N + j] = i;
               b[i * N + j] = i;
   }
```

Experiment:8

Aim: - Create, compile and run an a CUDA " Matrix multiplication **" program using remote access over server.**

Program:

```
/* ------------- COMPUTATION DONE ON GPU ----------------------------*/

   cudaMalloc((void**)&dev_a, size);                        // allocate memory on device
   cudaMalloc((void**)&dev_b, size);
   cudaMalloc((void**)&dev_c, size);

   cudaMemcpy(dev_a, a , size ,cudaMemcpyHostToDevice);
   cudaMemcpy(dev_b, b , size ,cudaMemcpyHostToDevice);
   cudaMemcpy(dev_c, c , size ,cudaMemcpyHostToDevice);

   cudaEventCreate(&start);                         // instrument code to measure start time
   cudaEventCreate(&stop);

   cudaEventRecord(start, 0);
//  cudaEventSynchronize(start);       // Needed?

   gpu_matrixadd<<<Grid,Block>>>(dev_a,dev_b,dev_c,N);

   cudaMemcpy(c,dev_c, size ,cudaMemcpyDeviceToHost);

   cudaEventRecord(stop, 0);          // instrument code to measue end time
   cudaEventSynchronize(stop);
   cudaEventElapsedTime(&elapsed_time_ms, start, stop );

//  for(i=0;i < N;i++)
//  for(j=0;j < N;j++)
//     printf("%d+%d=%d\n",a[i * N + j],b[i * N + j],c[i * N + j]);

   printf("Time to calculate results on GPU: %f ms.\n", elapsed_time_ms);  // print out execution time

/* ------------- COMPUTATION DONE ON HOST CPU ----------------------------*/

   cudaEventRecord(start, 0);                       // use same timing
//  cudaEventSynchronize(start);       // Needed?

   cpu_matrixadd(a,b,d,N);                          // do calculation on host

   cudaEventRecord(stop, 0);          // instrument code to measue end time
   cudaEventSynchronize(stop);
   cudaEventElapsedTime(&elapsed_time_ms, start, stop );

   printf("Time to calculate results on CPU: %f ms.\n", elapsed_time_ms);  // print out execution time

/* ------------------- check device creates correct results -----------------*/

   for(i=0;i < N*N;i++) {
                if (c[i] != d[i]) printf("********** ERROR in results, CPU and GPU create different answers
********\n");
```

```
                break;
    }

    printf("\nEnter c to repeat, return to terminate\n");
    scanf("%c",&key);
    scanf("%c",&key);

} while (key == 'c'); // loop of complete program

/* --------------  clean up  ---------------------------------------*/
    free(a);
    free(b);
    free(c);
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);

    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    return 0;
}
```

Flow of Program : Open text editor (like vi/vim) open a new file - call it whatever you'd like. It should do the following:

- Use the appropriate .cu include file
- Enter input parameters and data
- Computation done on GPU
- Computation done on CPU
- Call function in main
- Write connection termination segment
- Compile and this with: **$ nvcc**
- Print a general value of multiplication and execute task rank and processor name **Execution on GPU equipped server$ ./a.out**

Terminate the Connection environment