

Practical 1

Objective:

Identify the basic objectives and outcomes of parallel computing.

Theory:

Parallel computing is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel"). There are several different forms of parallel computing: bit-level, instruction level, data, and task parallelism.

Parallelism has been employed for many years, mainly in high-performance computing, but interest in it has grown lately due to the physical constraints preventing frequency scaling.

As power consumption (and consequently heat generation) by computers has become a concern in recent years, parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multi-core processors.

Objectives –

1. To learn how to design parallel programs and how to evaluate their execution.
2. To understand the characteristics, the benefits and the limitations of parallel systems.
3. To write implementation code in different parallel programming environments.

Outcomes –

1. Be able to reason about ways to parallelize a problem.
2. Be able to evaluate a parallel platform for a given problem.
3. Become familiar with programming with OpenMP, MPI and CUDA.

Conclusion:

Thus we have acquainted ourselves with the basics of parallel computing and identified the objectives to be accomplished and the outcomes to be generated.

Name of Assessing Faculty: Prof. S.T. Shingade

Signature: _____

Date: _____

OpenMP Programming

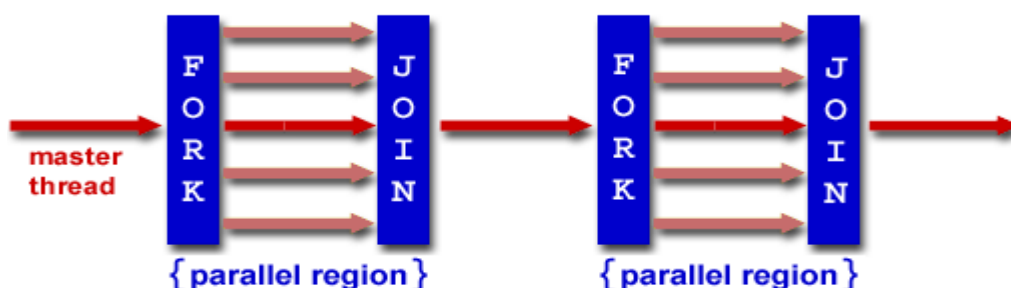
Practical 2a

Objective:

To write an OpenMP program for illustrating the Fork Join model.

Theory:

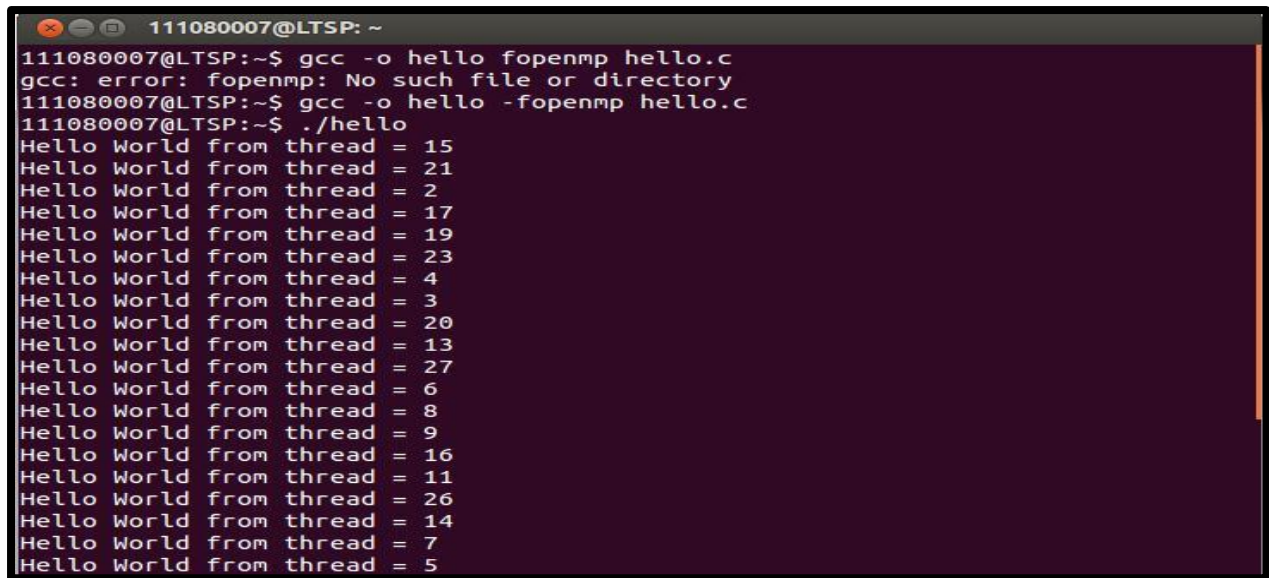
- OpenMP (Open Multi-Processing) is an API that supports multi-platform shared memory multiprocessing programming in C, C++ and Fortran on most processor architectures and operating systems, including Solaris, AIX, HP-UX, Linux, Mac OS X, and Windows platforms.
- It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.
- OpenMP is an implementation of multithreading, a method of parallelizing whereby a master thread (a series of instructions executed consecutively) forks a specified number of slave threads and a task is divided among them.
- The threads then run concurrently, with the runtime environment allocating threads to different processors.
- The section of code that is meant to run in parallel is marked accordingly, with a preprocessor directive that will cause the threads to form before the section is executed.
- Each thread has an id attached to it which can be obtained using a function (called `omp_get_thread_num()`).
- The thread id is an integer, and the master thread has an id of 0.
- After the execution of the parallelized code, the threads join back into the master thread, which continues onward to the end of the program.
- By default, each thread executes the parallelized section of code independently.
- Work-sharing constructs can be used to divide a task among the threads so that each thread executes its allocated part of the code. Both task parallelism and data parallelism can be achieved using OpenMP in this way.
- The runtime environment allocates threads to processors depending on usage, machine load and other factors.
- The number of threads can be assigned by the runtime environment based on environment variables or in code using functions.
- The OpenMP functions are included in a header file labelled `omp.h` in C/C++.
- The output may also be garbled because of the race condition caused from the two threads sharing the standard output.



Program:

```
#include<stdio.h>
#include<omp.h>
int main(){
    int nthreads, tid;

    /* Fork a team of threads giving them their own copies of variables */
    #pragma omp parallel private(nthreads, tid)
    {
        /* Obtain thread number */
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);
        /* Only master thread does this */
        if (tid == 0)
        {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
    } /* All threads join master thread and disband */
}
```

Output:

```
111080007@LTSP: ~  
111080007@LTSP:~$ gcc -o hello fopenmp hello.c  
gcc: error: fopenmp: No such file or directory  
111080007@LTSP:~$ gcc -o hello -fopenmp hello.c  
111080007@LTSP:~$ ./hello  
Hello World from thread = 15  
Hello World from thread = 21  
Hello World from thread = 2  
Hello World from thread = 17  
Hello World from thread = 19  
Hello World from thread = 23  
Hello World from thread = 4  
Hello World from thread = 3  
Hello World from thread = 20  
Hello World from thread = 13  
Hello World from thread = 27  
Hello World from thread = 6  
Hello World from thread = 8  
Hello World from thread = 9  
Hello World from thread = 16  
Hello World from thread = 11  
Hello World from thread = 26  
Hello World from thread = 14  
Hello World from thread = 7  
Hello World from thread = 5
```

Conclusion:

Thus we have implemented a simple program in OpenMP using threads in C to illustrate the Fork Join model.

Name of Assessing Faculty: Prof. S.T. Shingade

Signature: _____

Date: _____

Practical 2b

Objective:

To write an OpenMP program for solving the Producer Consumer problem.

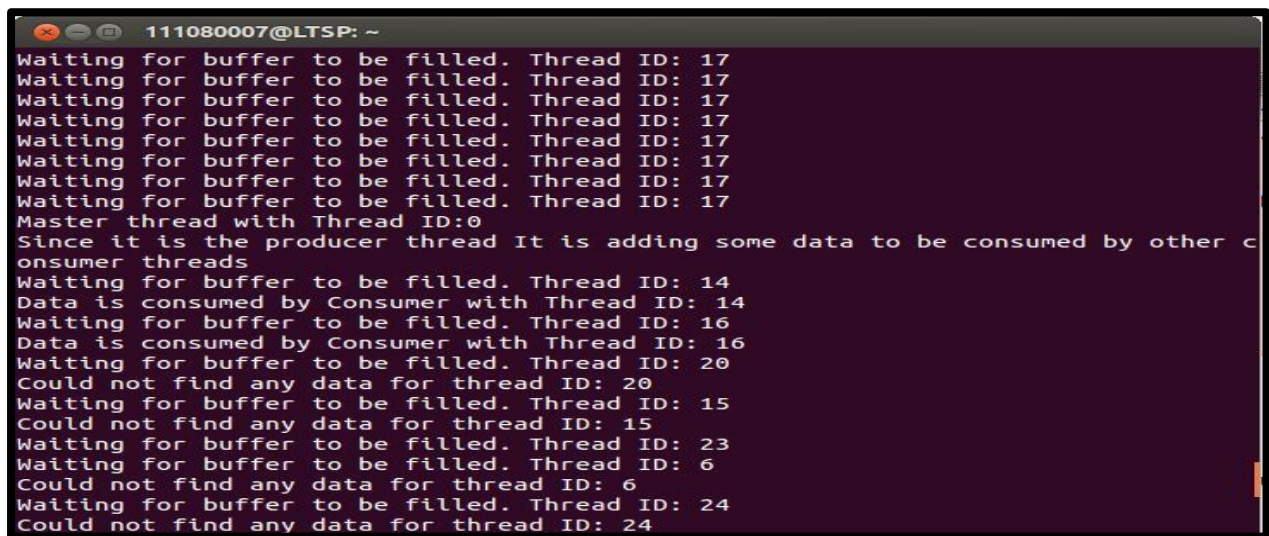
Theory:

In this program, the master thread will act as a producer while the other threads will wait until the master thread creates buffer, and once added the master notifies the threads using a shared variable and all other threads will consume the data.

Program:

```
#include<stdio.h>
#include<omp.h>
int main()
{
int i=0;
int x=0;
#pragma omp parallel shared(i)
{
if(omp_get_thread_num()==0)
{
printf("Master thread with Thread ID:%d\n", omp_get_thread_num());
printf("Since it is the producer thread It is adding some data to be consumed by other consumer threads\n");
i+=10;
x=1;
} else
{
while(x==0)
printf("Waiting for buffer to be filled. Thread ID: %d\n",omp_get_thread_num());
#pragma critical
{
if(i>0){
printf("Data is consumed by Consumer with Thread ID: %d\n",omp_get_thread_num());
```

```
i-=5;
} else {
printf("Could not find any data for thread ID: %d\n",omp_get_thread_num());
}}
}
}
}
```

Output:

```
111080007@LTSP: ~
Waiting for buffer to be filled. Thread ID: 17
Waiting for buffer to be filled. Thread ID: 17
Waiting for buffer to be filled. Thread ID: 17
Waiting for buffer to be filled. Thread ID: 17
Waiting for buffer to be filled. Thread ID: 17
Waiting for buffer to be filled. Thread ID: 17
Waiting for buffer to be filled. Thread ID: 17
Waiting for buffer to be filled. Thread ID: 17
Master thread with Thread ID:0
Since it is the producer thread It is adding some data to be consumed by other c
onsumer threads
Waiting for buffer to be filled. Thread ID: 14
Data is consumed by Consumer with Thread ID: 14
Waiting for buffer to be filled. Thread ID: 16
Data is consumed by Consumer with Thread ID: 16
Waiting for buffer to be filled. Thread ID: 20
Could not find any data for thread ID: 20
Waiting for buffer to be filled. Thread ID: 15
Could not find any data for thread ID: 15
Waiting for buffer to be filled. Thread ID: 23
Waiting for buffer to be filled. Thread ID: 6
Could not find any data for thread ID: 6
Waiting for buffer to be filled. Thread ID: 24
Could not find any data for thread ID: 24
```

Conclusion:

Thus, we have implemented and studied the producer consumer problem using OpenMP.

Name of Assessing Faculty: Prof. S.T. Shingade

Signature: _____

Date: _____

Practical 2c

Objective:

To write an OpenMP Program for Matrix Multiplication.

Program:

```
#include<stdio.h>
#include<omp.h>
int main() {
    int i,j,k,m,n,p;
    printf("Enter the number of rows in Matrix 1:");
    scanf("%d",&m);
    int *matrixA[m];
    printf("Enter the number of columns in Matrix 1:");
    scanf("%d",&n);
    for(i=0;i<m;i++){
        matrixA[i] = (int *)malloc(n*sizeof(int));
    }
    printf("<--Now Input the values for matrix 1 row-wise-->\n");
    for(i=0;i<m;i++){
        for(j=0;j<n;j++){
            scanf("%d",&matrixA[i][j]);
        }
    }
    printf("Enter the number of columns in Matrix 2:");
    scanf("%d",&p);
    int *matrixB[n];
    for(i=0;i<n;i++){
        matrixB[i] = (int *)malloc(p*sizeof(int));
    }
    printf("<--Now Input the values for matrix 2 row-wise-->\n");
    for(i=0;i<n;i++){
        for(j=0;j<p;j++){
```



```
scanf("%d",&matrixB[i][j]);
}
}
int matrixC[m][p];
#pragma omp parallel private(i,j,k) shared(matrixA,matrixB,matrixC)
{
#pragma omp for schedule(static)
for (i=0; i<m; i=i+1){
    for (j=0; j<p; j=j+1){
        matrixC[i][j] = 0;
        for (k=0; k<n; k=k+1){
            matrixC[i][j]=(matrixC[i][j])+((matrixA[i][k])*(matrixB[k][j]));
        }
    }
}
}
printf("The output after Matrix Multiplication is: \n");
for(i=0;i<m;i++){
    for(j=0;j<p;j++){
        printf("%d \t",matrixC[i][j]);
        printf("\n");
    }
}
return 0;
}
```

Output:

```
sid1793@ubuntu: ~  
sid1793@ubuntu:~$ gcc -o matrix -fopenmp matrixmult.c  
sid1793@ubuntu:~$ ./matrix 2 3 4 5 6  
  
Threads : 2  
Matrix A Size : 3 X 4  
Matrix B Size : 5 X 6  
Matrix Matrix Computation Is Not Possible  
sid1793@ubuntu:~$ ./matrix 2 3 4 4 6  
  
Threads : 2  
Matrix A Size : 3 X 4  
Matrix B Size : 4 X 6  
  
Matrix into Matrix Multiplication using Parallel for directive.  
.....Done  
  
Time in Seconds (T) : 0.000426 Seconds  
  
( T represents the Time taken for computation )  
.....  
.....  
sid1793@ubuntu:~$ █
```

Conclusion: Thus, I have implemented an OpenMP program for Matrix Multiplication.

Name of Assessing Faculty: Prof. S.T. Shingade

Signature: _____

Date: _____

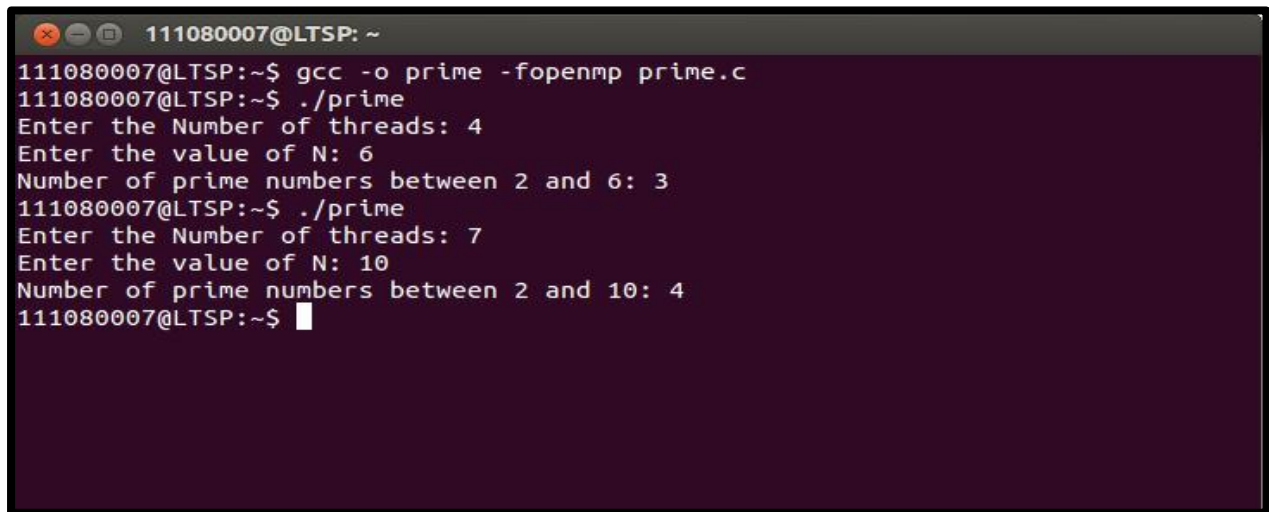
Practical 2d

Objective:

To write an OpenMP program to find prime numbers between 2 and the given number, and store all the prime numbers in an array.

Program:

```
#include<stdio.h>
#include<omp.h>
int IsPrime(int number) {
    int i;
    for (i = 2; i < number; i++) {
        if (number % i == 0 && i != number) return 0;
    }
    return 1;
}
int main() {
    int noOfThreads,valueN,indexCount=0,arrayVal[10000],tempValue;
    printf("Enter the Number of threads: ");
    scanf("%d",&noOfThreads);
    printf("Enter the value of N: ");
    scanf("%d",&valueN);
    omp_set_num_threads(noOfThreads);
    #pragma omp parallel for reduction(+:indexCount)
    for(tempValue=2;tempValue<=valueN;tempValue++){
        if(IsPrime(tempValue)){
            arrayVal[indexCount] = tempValue;
            indexCount++;
        }
    }
    printf("Number of prime numbers between 2 and %d: %d\n",valueN,indexCount);
    return 0;
}
```

Output:A terminal window with a dark purple background and white text. The window title is "111080007@LTSP: ~". The user enters the command "gcc -o prime -fopenmp prime.c" and presses enter. Then they enter "./prime" and press enter. The program prompts "Enter the Number of threads: 4", the user enters "4", and the program prompts "Enter the value of N: 6", the user enters "6". The program outputs "Number of prime numbers between 2 and 6: 3". Then the user enters "./prime" again. The program prompts "Enter the Number of threads: 7", the user enters "7", and the program prompts "Enter the value of N: 10", the user enters "10". The program outputs "Number of prime numbers between 2 and 10: 4". Finally, the user enters a blank line, and the prompt "111080007@LTSP:~\$" is shown with a cursor.

```
111080007@LTSP:~$ gcc -o prime -fopenmp prime.c
111080007@LTSP:~$ ./prime
Enter the Number of threads: 4
Enter the value of N: 6
Number of prime numbers between 2 and 6: 3
111080007@LTSP:~$ ./prime
Enter the Number of threads: 7
Enter the value of N: 10
Number of prime numbers between 2 and 10: 4
111080007@LTSP:~$
```

Conclusion:

Thus, I have implemented an OpenMP program for finding prime numbers between 2 and N.

Name of Assessing Faculty: Prof. S.T. Shingade

Signature: _____

Date: _____

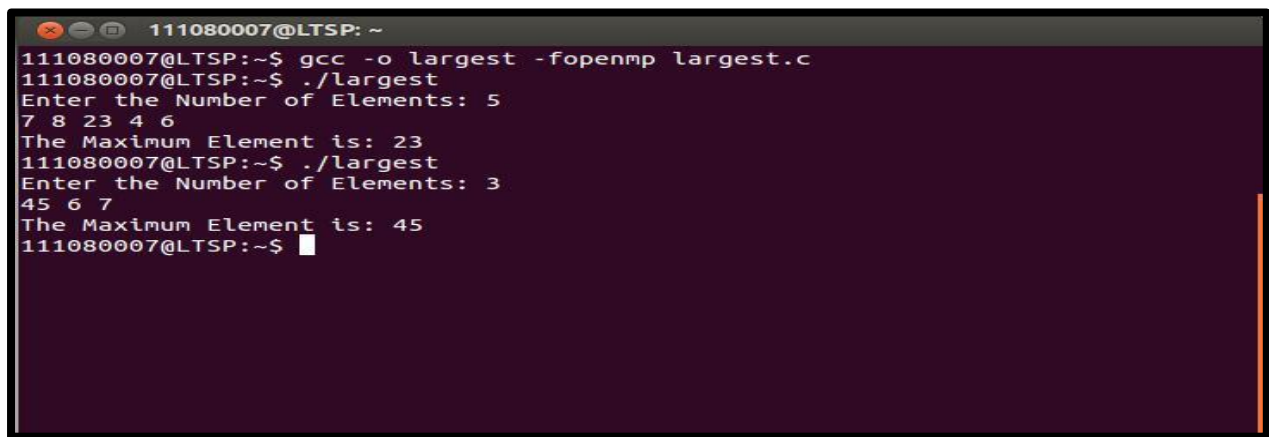
Practical 2e

Objective:

To write an OpenMP program to print the Largest Element in an array.

Program:

```
#include<stdio.h>
#include<omp.h>
int main() {
    int numberOfElements,currentMax=-1,iIterator,arrayInput[10000];
    printf("Enter the Number of Elements: ");
    scanf("%d",&numberOfElements);
    for(iIterator=0;iIterator<numberOfElements;iIterator++){
        scanf("%d",&arrayInput[iIterator]);
    }
    #pragma omp parallel for shared(currentMax)
    for(iIterator=0;iIterator<numberOfElements;iIterator++){
        #pragma omp critical
        if(arrayInput[iIterator] > currentMax){
            currentMax = arrayInput[iIterator];
        }
    }
    printf("The Maximum Element is: %d\n",currentMax);
    return 0;
}
```

Output:A terminal window with a dark purple background and white text. The window title is "111080007@LTSP: ~". The user enters the command "gcc -o largest -fopenmp largest.c", followed by "./largest". The program prompts "Enter the Number of Elements: 5", and the user enters "7 8 23 4 6". The program outputs "The Maximum Element is: 23". The user then enters another command "./largest", and the program prompts "Enter the Number of Elements: 3", to which the user enters "45 6 7". The program outputs "The Maximum Element is: 45". The prompt "111080007@LTSP:~\$" is visible at the end of the line.

```
111080007@LTSP:~$ gcc -o largest -fopenmp largest.c
111080007@LTSP:~$ ./largest
Enter the Number of Elements: 5
7 8 23 4 6
The Maximum Element is: 23
111080007@LTSP:~$ ./largest
Enter the Number of Elements: 3
45 6 7
The Maximum Element is: 45
111080007@LTSP:~$
```

Conclusion:

Thus, I have implemented an OpenMP program for finding the largest element in an array.

Name of Assessing Faculty: Prof. S.T. Shingade

Signature: _____

Date: _____

Practical 2f

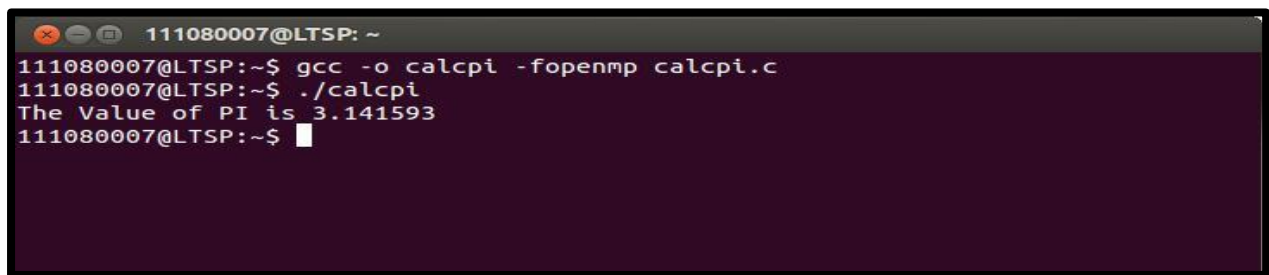
Objective:

To write an OpenMP program for Pi calculation.

Program:

```
#include<stdio.h>
#include<omp.h>
int main() {
int num_steps=10000,i;
double aux,pi,step = 1.0/(double) num_steps,x=0.0,sum = 0.0;
#pragma omp parallel private(i,x,aux) shared(sum)
{
#pragma omp for schedule(static)
for (i=0; i<num_steps; i=i+1){

x=(i+0.5)*step;
aux=4.0/(1.0+x*x);
#pragma omp critical
sum = sum + aux;
}
}
pi=step*sum;
printf("The Value of PI is %lf\n",pi);
return 0;
}
```

Output:A terminal window with a dark purple background and white text. The window title is "111080007@LTSP: ~". The text inside shows the following commands and output:

```
111080007@LTSP:~$ gcc -o calcp_i -fopenmp calcp_i.c
111080007@LTSP:~$ ./calcp_i
The Value of PI is 3.141593
111080007@LTSP:~$
```

Conclusion:

Thus, I have implemented an OpenMP program for calculating the value of Pi.

Name of Assessing Faculty: Prof. S.T. Shingade

Signature: _____

Date: _____

MPI Programming

Practical 3a

Objective:

To write a simple MPI program for calculating Rank and Number of processor.

Program:

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

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

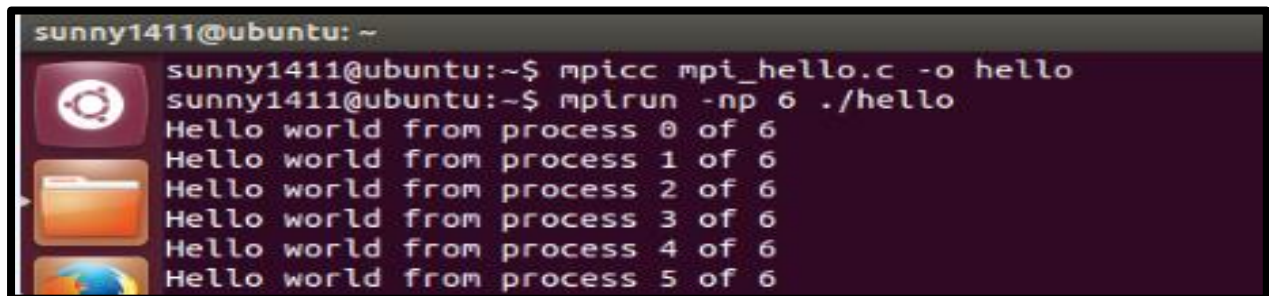
    MPI_Init (&argc, &argv);    /* starts MPI */

    MPI_Comm_rank (MPI_COMM_WORLD, &rank);    /* get current process id */
    MPI_Comm_size (MPI_COMM_WORLD, &size);    /* get number of processes */

    printf( "Hello world from process %d of %d\n", rank, size );

    MPI_Finalize();

    return 0;
}
```

Output:

```
sunny1411@ubuntu: ~
sunny1411@ubuntu:~$ mpicc mpi_hello.c -o hello
sunny1411@ubuntu:~$ mpirun -np 6 ./hello
Hello world from process 0 of 6
Hello world from process 1 of 6
Hello world from process 2 of 6
Hello world from process 3 of 6
Hello world from process 4 of 6
Hello world from process 5 of 6
```

Conclusion:

Thus, I have implemented MPI program for calculating rank and number of processors.

Name of Assessing Faculty: Prof. S.T. Shingade

Signature: _____

Date: _____

Practical 3b

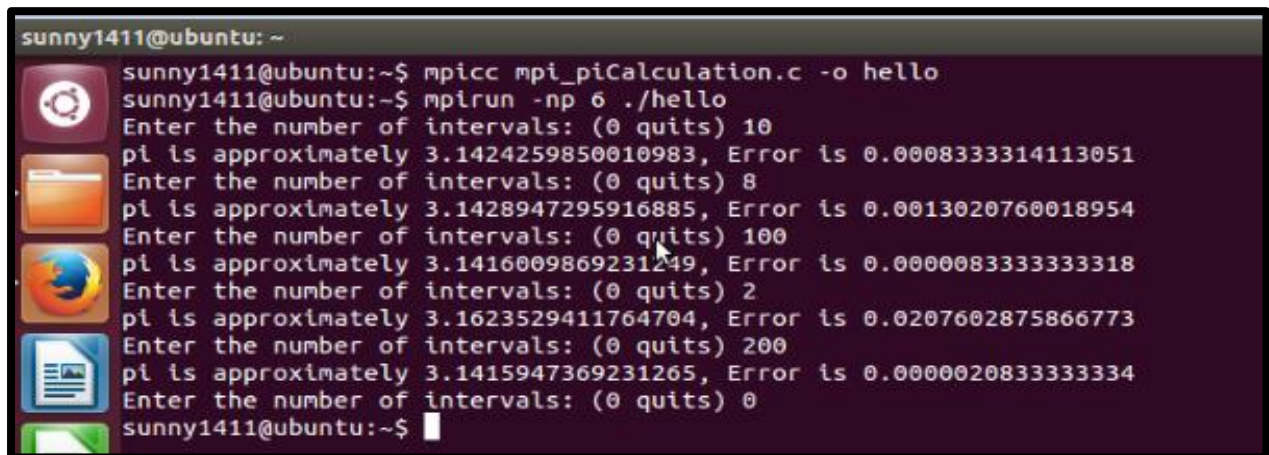
Objective:

To write an MPI program for Pi calculation.

Program:

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
int main( int argc, char *argv[] )
{
    int n, myid, numprocs, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    while (1) {
        if (myid == 0) {
            printf("Enter the number of intervals: (0 quits) ");
            scanf("%d",&n);
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0)
            break;
        else {
            h = 1.0 / (double) n;
            sum = 0.0;
            for (i = myid + 1; i <= n; i += numprocs) {
                x = h * ((double)i - 0.5);
                sum += (4.0 / (1.0 + x*x));
            }
            mypi = h * sum;
```

```
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
           MPI_COMM_WORLD);
if (myid == 0)
    printf("pi is approximately %.16f, Error is %.16f\n",
           pi, fabs(pi - PI25DT));
}
}
MPI_Finalize();
return 0;
}
```

Output:

```
sunny1411@ubuntu: ~
sunny1411@ubuntu:~$ mpicc mpi_piCalculation.c -o hello
sunny1411@ubuntu:~$ mpirun -np 6 ./hello
Enter the number of intervals: (0 quits) 10
pi is approximately 3.1424259850010983, Error is 0.0008333314113051
Enter the number of intervals: (0 quits) 8
pi is approximately 3.1428947295916885, Error is 0.0013020760018954
Enter the number of intervals: (0 quits) 100
pi is approximately 3.1416009869231249, Error is 0.0000083333333318
Enter the number of intervals: (0 quits) 2
pi is approximately 3.1623529411764704, Error is 0.0207602875866773
Enter the number of intervals: (0 quits) 200
pi is approximately 3.1415947369231265, Error is 0.0000020833333334
Enter the number of intervals: (0 quits) 0
sunny1411@ubuntu:~$
```

Conclusion:

Thus, I have implemented an MPI program for calculating the value of Pi.

Name of Assessing Faculty: Prof. S.T. Shingade

Signature: _____

Date: _____

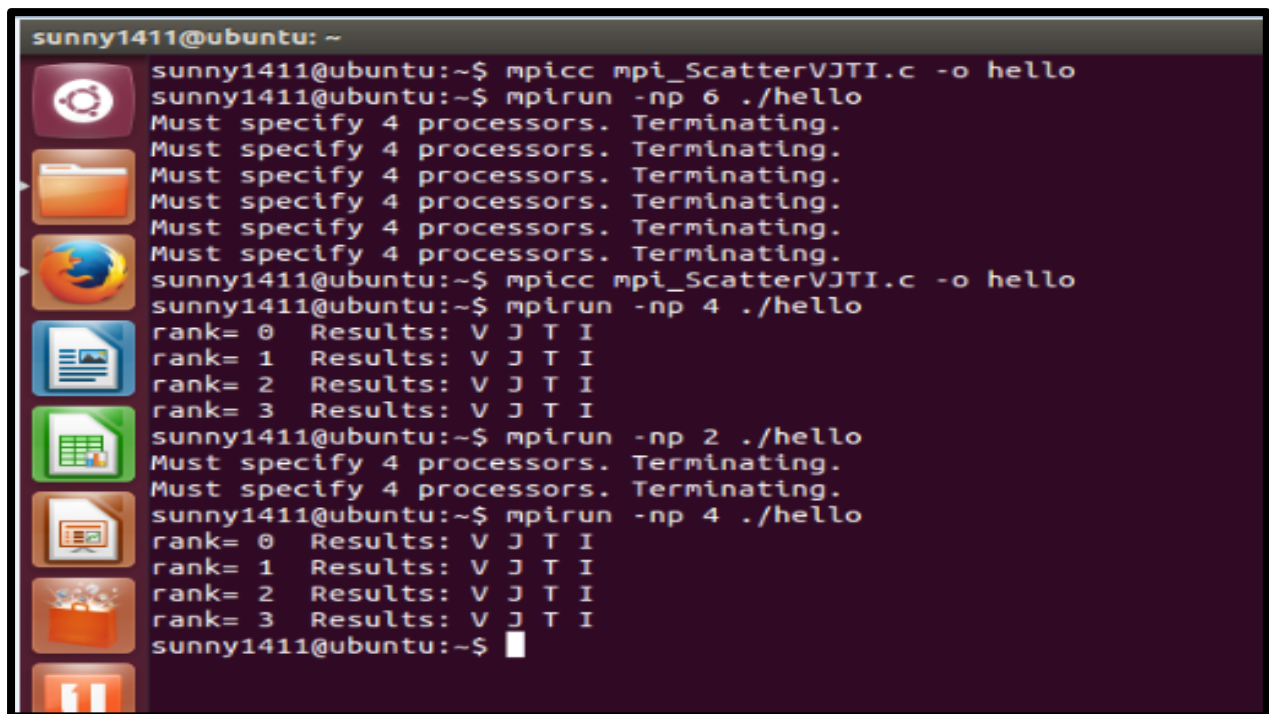
Practical 3c

Objective:

To write an advanced MPI program that has a total number of 4 processes, where the process with rank = 0 should send VJTI letter to all the processes using MPI_Scatter call.

Program:

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define SIZE 4
int main (int argc, char *argv[]) {
    int numtasks, rank, sendcount, recvcount, source;
    char sendbuf[SIZE][SIZE] = {
        {'V','J','T','T'},
        {'V','J','T','T'},
        {'V','J','T','T'},
        {'V','J','T','T'}};
    char recvbuf[SIZE];
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    if (numtasks == SIZE) {
        source = 0;
        sendcount = SIZE;
        recvcount = SIZE;
        MPI_Scatter(sendbuf,sendcount,MPI_CHAR,recvbuf,recvcount,
            MPI_CHAR,source,MPI_COMM_WORLD);
        printf("rank= %d Results: %c %c %c %c\n",rank,recvbuf[0],
            recvbuf[1],recvbuf[2],recvbuf[3]); }
    else
        printf("Must specify %d processors. Terminating.\n",SIZE);
    MPI_Finalize();
}
```

Output:A terminal window titled 'sunny1411@ubuntu: ~' with a sidebar of application icons. The terminal shows the following commands and output:

```
sunny1411@ubuntu:~$ mpicc mpi_ScatterVJTI.c -o hello
sunny1411@ubuntu:~$ mpirun -np 6 ./hello
Must specify 4 processors. Terminating.
Must specify 4 processors. Terminating.
Must specify 4 processors. Terminating.
Must specify 4 processors. Terminating.
Must specify 4 processors. Terminating.
sunny1411@ubuntu:~$ mpicc mpi_ScatterVJTI.c -o hello
sunny1411@ubuntu:~$ mpirun -np 4 ./hello
rank= 0 Results: V J T I
rank= 1 Results: V J T I
rank= 2 Results: V J T I
rank= 3 Results: V J T I
sunny1411@ubuntu:~$ mpirun -np 2 ./hello
Must specify 4 processors. Terminating.
Must specify 4 processors. Terminating.
sunny1411@ubuntu:~$ mpirun -np 4 ./hello
rank= 0 Results: V J T I
rank= 1 Results: V J T I
rank= 2 Results: V J T I
rank= 3 Results: V J T I
sunny1411@ubuntu:~$
```

Conclusion:

Thus, I have implemented an advanced MPI program for scattering “VJTI” to all the processes from the root process using MPI_Scatter Call.

Name of Assessing Faculty: Prof. S.T. Shingade

Signature: _____

Date: _____

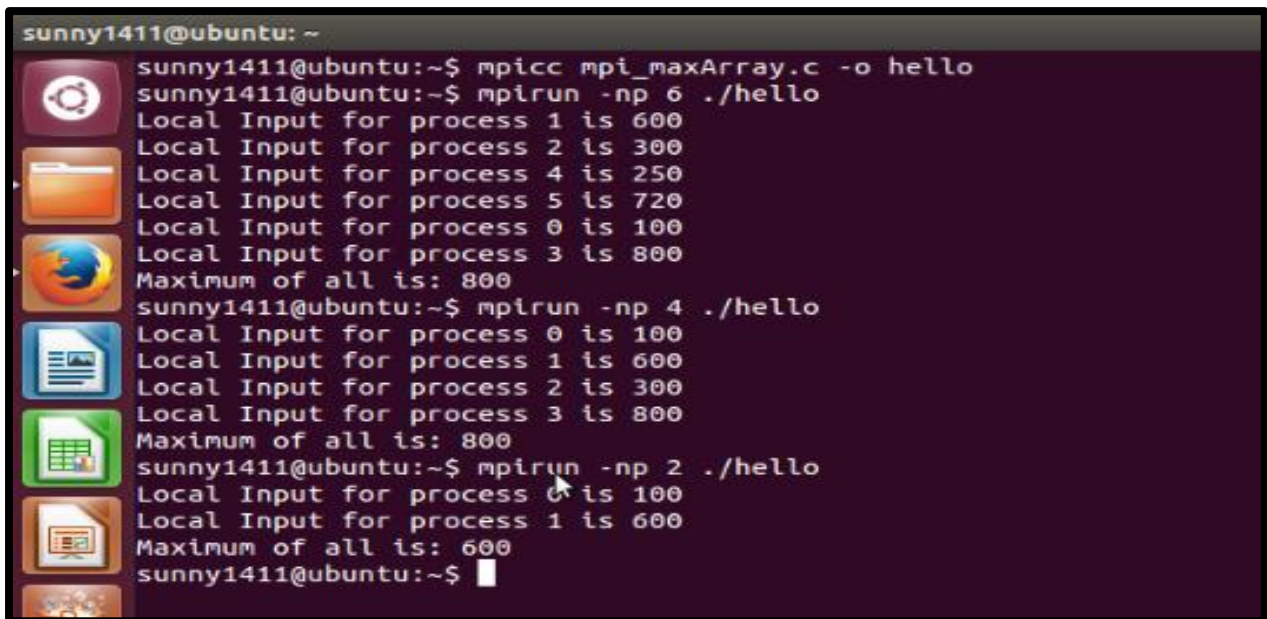
Practical 3d

Objective:

To write an advanced MPI program to find the maximum value in array of six integers with 6 processes, and print the result in root process using MPI_Reduce call.

Program:

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define SIZE 4
int main (int argc, char *argv[])
{
    int rank,numtasks,array[6] = {100,600,300,800,250,720},i,inputNumber;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    printf("Local Input for process %d is %d\n",rank,array[rank]);
    inputNumber = array[rank];
    int maxNumber;
    MPI_Reduce(&inputNumber, &maxNumber, 1, MPI_INT, MPI_MAX, 0,
              MPI_COMM_WORLD);
    // Print the result
    if (rank == 0) {
        printf("Maximum of all is: %d\n",maxNumber);
    }
    MPI_Finalize();
}
```

Output:A terminal window titled 'sunny1411@ubuntu: ~' showing the execution of an MPI program. The user first compiles 'mpi_maxArray.c' with 'mpicc' and then runs it with 'mpirun -np 6 ./hello'. The output shows local inputs for 6 processes (100, 600, 300, 250, 720, 800) and the maximum value of 800. This is repeated for 4 and then 2 processes, with the maximum value remaining 800. The terminal has a dark background with a sidebar of application icons on the left.

```
sunny1411@ubuntu: ~  
sunny1411@ubuntu:~$ mpicc mpi_maxArray.c -o hello  
sunny1411@ubuntu:~$ mpirun -np 6 ./hello  
Local Input for process 1 is 600  
Local Input for process 2 is 300  
Local Input for process 4 is 250  
Local Input for process 5 is 720  
Local Input for process 0 is 100  
Local Input for process 3 is 800  
Maximum of all is: 800  
sunny1411@ubuntu:~$ mpirun -np 4 ./hello  
Local Input for process 0 is 100  
Local Input for process 1 is 600  
Local Input for process 2 is 300  
Local Input for process 3 is 800  
Maximum of all is: 800  
sunny1411@ubuntu:~$ mpirun -np 2 ./hello  
Local Input for process 0 is 100  
Local Input for process 1 is 600  
Maximum of all is: 600  
sunny1411@ubuntu:~$
```

Conclusion:

Thus, I have implemented an advanced MPI program for finding the maximum of all the elements in an array of 6 elements using 6 processes using the MPI_Reduce Call.

Name of Assessing Faculty: Prof. S.T. Shingade

Signature: _____

Date: _____

Practical 3e

Objective:

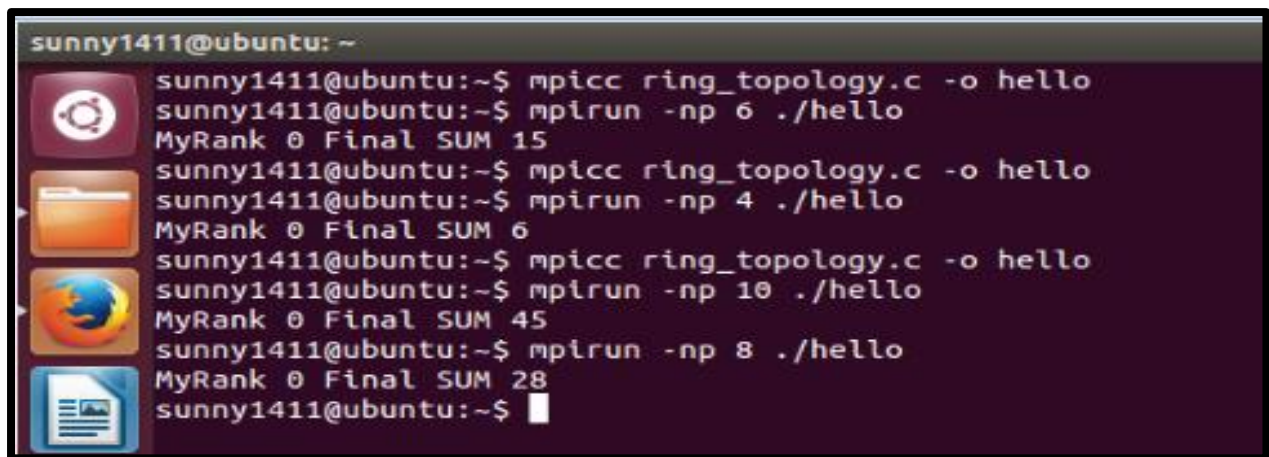
To write an MPI program for Ring topology.

Program:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{
    int    MyRank, Numprocs, Root = 0;
    int    value, sum = 0;
    int    Source, Source_tag;
    int    Destination, Destination_tag;
    MPI_Status status;

    /* Initialize MPI */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &Numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);
    if (MyRank == Root) {
        Destination = MyRank + 1;
        Destination_tag = 0;
        MPI_Send(&MyRank, 1, MPI_INT, Destination, Destination_tag,
                MPI_COMM_WORLD);
    }
    else {
        if (MyRank < Numprocs - 1) {
            Source = MyRank - 1;
            Source_tag = 0;
            MPI_Recv(&value, 1, MPI_INT, Source, Source_tag,
                    MPI_COMM_WORLD, &status);
            sum = MyRank + value;
            Destination = MyRank + 1;
```

```
    Destination_tag = 0;
    MPI_Send(&sum, 1, MPI_INT, Destination, Destination_tag,
             MPI_COMM_WORLD);
}
else{
    Source    = MyRank - 1;
    Source_tag = 0;
    MPI_Recv(&value, 1, MPI_INT, Source, Source_tag,
             MPI_COMM_WORLD, &status);
    sum = MyRank + value;
}
}
if (MyRank == Root)
{
    Source    = Numprocs - 1;
    Source_tag = 0;
    MPI_Recv(&sum, 1, MPI_INT, Source, Source_tag,
             MPI_COMM_WORLD, &status);
    printf("MyRank %d Final SUM %d\n", MyRank, sum);
}
if(MyRank == (Numprocs - 1)){
    Destination    = 0;
    Destination_tag = 0;
    MPI_Send(&sum, 1, MPI_INT, Destination, Destination_tag,
             MPI_COMM_WORLD);
}
MPI_Finalize();
}
```

Output:A terminal window with a dark purple background and light blue text. The prompt is 'sunny1411@ubuntu: ~'. The user enters 'mpicc ring_topology.c -o hello' and then runs it with 'mpirun -np 6 ./hello', which outputs 'MyRank 0 Final SUM 15'. This is repeated for 4, 10, and 8 processes, with outputs of 6, 45, and 28 respectively. The terminal has four icons on the left: a red circle with a white 'C', an orange folder, a blue and orange globe, and a blue document icon.

```
sunny1411@ubuntu: ~  
sunny1411@ubuntu:~$ mpicc ring_topology.c -o hello  
sunny1411@ubuntu:~$ mpirun -np 6 ./hello  
MyRank 0 Final SUM 15  
sunny1411@ubuntu:~$ mpicc ring_topology.c -o hello  
sunny1411@ubuntu:~$ mpirun -np 4 ./hello  
MyRank 0 Final SUM 6  
sunny1411@ubuntu:~$ mpicc ring_topology.c -o hello  
sunny1411@ubuntu:~$ mpirun -np 10 ./hello  
MyRank 0 Final SUM 45  
sunny1411@ubuntu:~$ mpirun -np 8 ./hello  
MyRank 0 Final SUM 28  
sunny1411@ubuntu:~$
```

Conclusion:

Thus, I have implemented an MPI program for ring topology.

Name of Assessing Faculty: Prof. S.T. Shingade

Signature: _____

Date: _____

Numerical Computing Programming

Practical 4a

Objective:

To write any one Numerical Computing program for implementing Trapezoid Rule with MPI.

Program:

```
#include <stdio.h>

/* We'll be using MPI routines, definitions, etc. */
#include <mpi.h>

void Get_data(int p, int my_rank, double* a_p, double* b_p, int* n_p);
double Trap(double local_a, double local_b, int local_n, double h); /* Calculate local area */
double f(double x); /* function we're integrating */
int main(int argc, char** argv) {
    int    my_rank; /* My process rank */
    int    p;      /* The number of processes */
    double a;      /* Left endpoint */
    double b;      /* Right endpoint */
    int    n;      /* Number of trapezoids */
    double h;      /* Trapezoid base length */
    double local_a; /* Left endpoint my process */
    double local_b; /* Right endpoint my process */
    int    local_n; /* Number of trapezoids for */
                /* my calculation */
    double my_area; /* Integral over my interval */
    double total;   /* Total area */
    int    source;  /* Process sending area */
    int    dest = 0; /* All messages go to 0 */
    int    tag = 0;
    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, &p);
Get_data(p, my_rank, &a, &b, &n);
h = (b-a)/n; /* h is the same for all processes */
local_n = n/p; /* So is the number of trapezoids */
/* Length of each process' interval of
 * integration = local_n*h. So my interval
 * starts at: */
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
my_area = Trap(local_a, local_b, local_n, h);
/* Add up the areas calculated by each process */
if (my_rank == 0) {
    total = my_area;
    for (source = 1; source < p; source++) {
        MPI_Recv(&my_area, 1, MPI_DOUBLE, source, tag,
            MPI_COMM_WORLD, &status);
        total = total + my_area;
    }
} else {
    MPI_Send(&my_area, 1, MPI_DOUBLE, dest,
        tag, MPI_COMM_WORLD);
}
/* Print the result */
if (my_rank == 0) {
    printf("With n = %d trapezoids, our estimate\n",
        n);
    printf("of the area from %f to %f = %.15f\n",
        a, b, total);
}
/* Shut down MPI */
MPI_Finalize();
return 0;
} /* main */

```

```

/*-----
* Function:   Get_data
* Purpose:    Read in the data on process 0 and send to other
*             processes
* Input args: p, my_rank
* Output args: a_p, b_p, n_p
*/
void Get_data(int p, int my_rank, double* a_p, double* b_p, int* n_p) {
    int    q;
    MPI_Status status;
    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
        for (q = 1; q < p; q++) {
            MPI_Send(a_p, 1, MPI_DOUBLE, q, 0, MPI_COMM_WORLD);
            MPI_Send(b_p, 1, MPI_DOUBLE, q, 0, MPI_COMM_WORLD);
            MPI_Send(n_p, 1, MPI_INT, q, 0, MPI_COMM_WORLD);
        }
    } else {
        MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
        MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
        MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    }
} /* Get_data */

/*-----
* Function:   Trap
* Purpose:    Estimate a definite area using the trapezoidal
*             rule
* Input args: local_a (my left endpoint)
*             local_b (my right endpoint)
*             local_n (my number of trapezoids)
*             h (stepsize = length of base of trapezoids)
* Return val: Trapezoidal rule estimate of area from

```

```

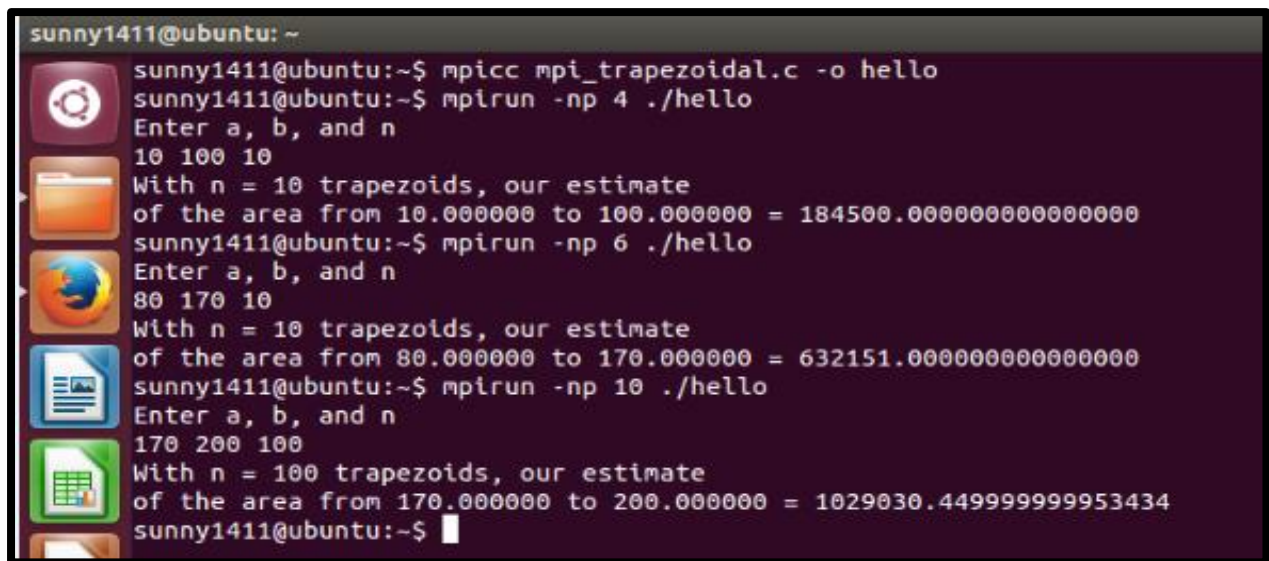
*          local_a to local_b
*/

double Trap(
    double local_a /* in */,
    double local_b /* in */,
    int    local_n /* in */,
    double h      /* in */) {
    double my_area; /* Store my result in my_area */
    double x;
    int i;
    my_area = (f(local_a) + f(local_b))/2.0;
    x = local_a;
    for (i = 1; i <= local_n-1; i++) {
        x = local_a + i*h;
        my_area = my_area + f(x);
    }
    my_area = my_area*h;
    return my_area;
} /* Trap */

/*-----
* Function:  f
* Purpose:   Compute value of function to be integrated
* Input args: x
*/

double f(double x) {
    double return_val;
    return_val = x*x + 1.0;
    return return_val;
} /* f */

```


Output:A terminal window with a dark purple background and light blue text. The prompt is 'sunny1411@ubuntu: ~'. The user enters 'mpicc mpi_trapezoidal.c -o hello'. Then 'mpirun -np 4 ./hello'. The program prompts 'Enter a, b, and n' and the user enters '10 100 10'. The program outputs 'With n = 10 trapezoids, our estimate of the area from 10.000000 to 100.000000 = 184500.0000000000000000'. Then the user enters 'mpirun -np 6 ./hello'. The program prompts 'Enter a, b, and n' and the user enters '80 170 10'. The program outputs 'With n = 10 trapezoids, our estimate of the area from 80.000000 to 170.000000 = 632151.0000000000000000'. Then the user enters 'mpirun -np 10 ./hello'. The program prompts 'Enter a, b, and n' and the user enters '170 200 100'. The program outputs 'With n = 100 trapezoids, our estimate of the area from 170.000000 to 200.000000 = 1029030.449999999953434'. The prompt 'sunny1411@ubuntu:~\$' is visible at the bottom.

```
sunny1411@ubuntu: ~  
sunny1411@ubuntu:~$ mpicc mpi_trapezoidal.c -o hello  
sunny1411@ubuntu:~$ mpirun -np 4 ./hello  
Enter a, b, and n  
10 100 10  
With n = 10 trapezoids, our estimate  
of the area from 10.000000 to 100.000000 = 184500.0000000000000000  
sunny1411@ubuntu:~$ mpirun -np 6 ./hello  
Enter a, b, and n  
80 170 10  
With n = 10 trapezoids, our estimate  
of the area from 80.000000 to 170.000000 = 632151.0000000000000000  
sunny1411@ubuntu:~$ mpirun -np 10 ./hello  
Enter a, b, and n  
170 200 100  
With n = 100 trapezoids, our estimate  
of the area from 170.000000 to 200.000000 = 1029030.449999999953434  
sunny1411@ubuntu:~$
```

Conclusion:

Thus, I have implemented MPI program for Trapezoidal Rule as a Numerical Computing problem.

Name of Assessing Faculty: Prof. S.T. Shingade

Signature: _____

Date: _____

Practical 4b

Objective:

To write any one Numerical Computing program for implementing Gaussian Filter with MPI.

Theory:

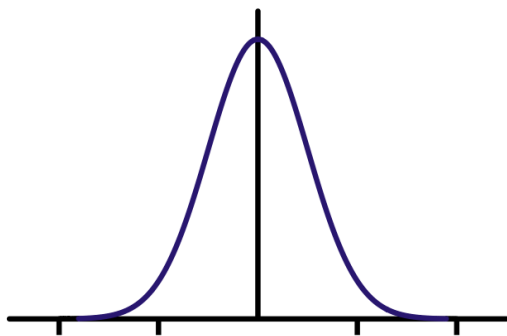
In electronics and signal processing, a Gaussian filter is a filter whose impulse response is a Gaussian function (or an approximation to it). Gaussian filters have the properties of having no overshoot to a step function input while minimizing the rise and fall time. This behavior is closely connected to the fact that the Gaussian filter has the minimum possible group delay. It is considered the ideal time domain filter, just as the sinc is the ideal frequency domain filter. These properties are important in areas such as oscilloscopes and digital telecommunication systems.

In two dimensions, it is the product of two such Gaussians, one per direction-

$$g(x, y) = \frac{1}{2\pi\sigma^2} \cdot e^{-\frac{x^2+y^2}{2\sigma^2}}$$

where x is the distance from the origin in the horizontal axis, y is the distance from the origin in the vertical axis, and σ is the standard deviation of the Gaussian distribution.

Shape of the impulse response of a typical Gaussian filter-



Conclusion:

Thus, I have studied about Gaussian filter, theoretically.

Name of Assessing Faculty: Prof. S.T. Shingade

Signature: _____

Date: _____

CUDA Programming

Practical 5a

Objective:

To write a simple CUDA Program for ‘Hello World’.

Theory:

'CUDA ' stands for Compute Unified Device Architecture, it is a parallel computing platform and programming model created by NVIDIA and implemented by the graphics processing units (GPUs) that they produce. CUDA gives developers direct access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs.

Using CUDA, the GPUs can be used for general purpose processing (i.e., not exclusively graphics); this approach is known as GPGPU. Unlike CPUs, however, GPUs have a parallel throughput architecture that emphasizes executing many concurrent threads slowly, rather than executing a single thread very quickly.

Program:

```
#include <stdio.h>
#include <cuda.h>
#include <sys/time.h>
#include <assert.h>
__global__ void kernel (void) {}
int main(void) {
    kernel<<input parameters>>();
    printf("Hello, World");
    return 0;
}
```

Conclusion:

Thus, I have studied basics of CUDA programming and implemented the ‘Hello World’ program.

Name of Assessing Faculty: Prof. S.T. Shingade

Signature: _____

Date: _____

Practical 5b

Objective:

To write a CUDA program for Matrix Addition.

Program:

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <string.h>
#include <cuda.h>
#include <assert.h>

const int N = 4;
const int blocksize = 2;

__global__ void add_matrix_on_gpu( float* a, float *b, float *c, int N )
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i + j*N;
    if ( i < N && j < N )
        c[index] = a[index] + b[index];
}

void add_matrix_on_cpu(float *a, float *b, float *d)
{
    int i;
    for(i = 0; i < N*N; i++)
        d[i] = a[i]+b[i];
}

int main()
{
    float *a = new float[N*N];
    float *b = new float[N*N];
    float *c = new float[N*N];
```

```

float *d = new float[N*N];
for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f; b[i] = 3.5f; }
/*
printf("Matrix A:\n");
for(int i=0; i<N*N; i++)
{
    printf("\t%f",a[i]);
    if((i+1)%N==0)
        printf("\n");
}
printf("Matrix B:\n");
for(int i=0; i<N*N; i++)
{
    printf("\t%f",b[i]);
    if((i+1)%N==0)
        printf("\n");
}
*/

struct timeval  TimeValue_Start;
struct timezone TimeZone_Start;
struct timeval  TimeValue_Final;
struct timezone TimeZone_Final;
long           time_start, time_end;
double         time_overhead;

float *ad, *bd, *cd;
const int size = N*N*sizeof(float);
cudaMalloc( (void**)&ad, size );
cudaMalloc( (void**)&bd, size );
cudaMalloc( (void**)&cd, size );
cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );
dim3 dimBlock( blocksize, blocksize );
dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );

```

```

    gettimeofday(&TimeValue_Start, &TimeZone_Start);
    add_matrix_on_gpu<<<dimGrid, dimBlock>>>( ad, bd, cd, N );
    gettimeofday(&TimeValue_Final, &TimeZone_Final);
    cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );
    add_matrix_on_cpu(a,b,d);
    time_end = TimeValue_Final.tv_sec * 1000000 + TimeValue_Final.tv_usec;
    time_start = TimeValue_Start.tv_sec * 1000000 + TimeValue_Start.tv_usec;
    time_overhead = (time_end - time_start)/1000000.0;
/*    printf("result is:\n");
    for(int i=0; i<N*N; i++)
    {
        printf("\t%f%f",c[i],d[i]);
        if((i+1)%N==0)
            printf("\n");
    }
*/
    for(int i=0; i<N*N; i++)
        assert(c[i]==d[i]);
    printf("\n\t\tTime in Seconds (T)      : %lf\n\n",time_overhead);
    cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
    delete[] a; delete[] b; delete[] c; delete[] d;
    return EXIT_SUCCESS;
}

```

Conclusion:

Thus, I implemented a CUDA program for Matrix Addition.

Name of Assessing Faculty: Prof. S.T. Shingade

Signature: _____

Date: _____

Practical 5c

Objective:

To write a CUDA program for Prefix Sum.

Program:

```
#include<stdio.h>
#include<cuda.h>
#include <assert.h>
#include<sys/time.h>
#define N 5
#define BLOCKSIZE 5
__global__ void PrefixSum(float *dInArray, float *dOutArray, int arrayLen, int threadDim)
{
    int tidx = threadIdx.x;
    int tidy = threadIdx.y;
    int tindex = (threadDim * tidx) + tidy;
    int maxNumThread = threadDim * threadDim;
    int pass = 0;
    int count ;
    int curEleInd;
    float tempResult = 0.0;
    while( (curEleInd = (tindex + maxNumThread * pass)) < arrayLen )
    {
        tempResult = 0.0f;
        for( count = 0; count <= curEleInd; count++)
            tempResult += dInArray[count];
        dOutArray[curEleInd] = tempResult;
        pass++;
    }
    __syncthreads();
} //end of Prefix sum function
```



```

void PrefixSum_cpu(float *x_h, float *z_h)
{
    int i;
    for(i=0; i<N; i++)
    {
        if(i==0)
            z_h[i]=x_h[i];
        else
            z_h[i]=z_h[i-1]+x_h[i];
    }
}

int main()
{
    float *x_h, *y_h, *z_h;
    float *x_d, *y_d;
    int i;
    struct timeval TimeValue_Start;
    struct timezone TimeZone_Start;
    struct timeval TimeValue_Final;
    struct timezone TimeZone_Final;
    long time_start, time_end;
    double time_overhead;
    size_t size = N*sizeof(float);
    x_h = (float *)malloc(size);
    y_h = (float *)malloc(size);
    z_h = (float *)malloc(size);
    cudaMalloc((void **)&x_d,size);
    cudaMalloc((void **)&y_d,size);
    for(i=0; i<N; i++)
    {
        x_h[i] = (float) i+1;
    }
    cudaMemcpy(x_d,x_h,size,cudaMemcpyHostToDevice);
    dim3 dimBlock(BLOCKSIZE,BLOCKSIZE);

```

```

    dim3 dimGrid(1,1);
    gettimeofday(&TimeValue_Start, &TimeZone_Start);
    PrefixSum<<<dimGrid, dimBlock>>>(x_d, y_d, N, BLOCKSIZE);
    gettimeofday(&TimeValue_Final, &TimeZone_Final);
    cudaMemcpy(y_h,y_d,size,cudaMemcpyDeviceToHost);
    PrefixSum_cpu(x_h,z_h);
    for(i = 0; i < N; i++)
        assert(y_h[i]==z_h[i]);
    time_end = TimeValue_Final.tv_sec * 1000000 + TimeValue_Final.tv_usec;
    time_start = TimeValue_Start.tv_sec * 1000000 + TimeValue_Start.tv_usec;
    time_overhead = (time_end - time_start)/1000000.0;
    printf("\n\t\t Time in Seconds (T)      : %lf\n\n",time_overhead);
    free(x_h);
    free(y_h);
    free(z_h);
    cudaFree(x_d);
    cudaFree(y_d);
return 0;
}

```

Conclusion:

Thus, I have implemented a CUDA program for Prefix Sum.

Name of Assessing Faculty: Prof. S.T. Shingade

Signature: _____

Date: _____

Practical 5d

Objective:

To write a CUDA program for Matrix Transpose.

Program:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <cuda.h>
#include <assert.h>
#include <sys/time.h>

const int N = 8;
const int blocksize = 4;

__global__ void transpose_naive( float *out, float *in, const int N ) {
    unsigned int xIdx = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int yIdx = blockDim.y * blockIdx.y + threadIdx.y;
    if ( xIdx < N && yIdx < N ) {
        unsigned int idx_in = xIdx + N * yIdx;
        unsigned int idx_out = yIdx + N * xIdx;
        out[idx_out] = in[idx_in];
    }
}

void mat_trans_cpu(float *a, float *c)
{
    int mn  = N*N;    /* N rows and N columns */
    int q   = mn - 1;
    int i = 0;    /* Index of 1D array that represents the matrix */
    do
    {
        int k = (i*N) % q;
        while (k>i)
            k = (N*k) % q;
```

```

        if (k!=i)
        {
            c[k] = a[i];
            c[i] = a[k];
        }
        else
            c[i] = a[i];
    } while ( ++i <= (mn -2) );
c[i]=a[i];
    /* Update row and column */
/*    matrix.M = N;
    matrix.N = M;*/
}
int main()
{
    float *a = new float[N*N];
    float *b = new float[N*N];
    float *c = new float[N*N];
    int i;
    for ( i = 0; i < N*N; ++i ) {
        a[i] = drand48();
    }
    struct timeval TimeValue_Start;
    struct timezone TimeZone_Start;
    struct timeval TimeValue_Final;
    struct timezone TimeZone_Final;
    long        time_start, time_end;
    double       time_overhead;
/*
    for ( i = 0; i < N*N; i++)
    {
        printf("\t%f",a[i]);
        if(((i+1)%N == 0))
            printf("\n");
    }
}

```

```

    }
*/

float *ad, *bd ;
const int size = N*N*sizeof(float);
cudaMalloc( (void**)&ad, size );
cudaMalloc( (void**)&bd, size );
cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
dim3 dimBlock( blocksize, blocksize );
dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
gettimeofday(&TimeValue_Start, &TimeZone_Start);
transpose_naive<<<dimGrid, dimBlock>>>( bd, ad, N );
gettimeofday(&TimeValue_Final, &TimeZone_Final);
cudaMemcpy( b, bd, size, cudaMemcpyDeviceToHost );
mat_trans_cpu(a,c);
/* To display uncomment this section */
/*
printf("result matrix\n");
for ( i = 0; i < N*N; ++i ){
    printf("\t%f%f",b[i],c[i]);
    if( ((i+1)%N == 0))
        printf("\n");
}
*/

time_end = TimeValue_Final.tv_sec * 1000000 + TimeValue_Final.tv_usec;
time_start = TimeValue_Start.tv_sec * 1000000 + TimeValue_Start.tv_usec;
time_overhead = (time_end - time_start)/1000000.0;
for(i=0; i<N*N; i++)
    assert(b[i]==c[i]);
printf("\n\t\t Time in Seconds (T)      : %lf\n\n",time_overhead);
cudaFree( ad ); cudaFree( bd );
delete[] a; delete[] b, delete[] c;
return EXIT_SUCCESS;
}

```

Conclusion:

Thus, I have implemented a CUDA program for Matrix Transpose.

Name of Assessing Faculty: Prof. S.T. Shingade

Signature: _____

Date: _____

Practical 5e

Objective:

To write a CUDA program for Vector Addition

Program:

```
#include <stdio.h>
#include <cuda.h>
#include <stdlib.h>
#include <assert.h>
#include <sys/time.h>

#define N 4096 // size of array

__global__ void vectorAdd(int *a,int *b, int *c){
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if(tid < N)
        c[tid] = a[tid]+b[tid];
}

int main(int argc, char *argv[]) {
    int T = 10, B = 1; // threads per block and blocks per grid
    int a[N],b[N],c[N]; // vectors, statically declared
    int *dev_a, *dev_b, *dev_c;
    printf("Size of array = %d\n", N);
    do {
        printf("Enter number of threads per block (1024 max, comp. cap. 2.x ");
        scanf("%d",&T);
        printf("\nEnter number of blocks per grid: ");
        scanf("%d",&B);
        if (T * B < N) printf("Error T x B < N, try again\n");
    } while (T * B < N);
    cudaEvent_t start, stop; // using cuda events to measure time
    float elapsed_time_ms;
    cudaMalloc((void**)&dev_a,N * sizeof(int));
    cudaMalloc((void**)&dev_b,N * sizeof(int));
```

```

    cudaMalloc((void**)&dev_c,N * sizeof(int));
    for(int i=0;i<N;i++) { // load arrays with some numbers
        a[i] = i;
        b[i] = i*2;
    }
    cudaMemcpy(dev_a, a , N*sizeof(int),cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b , N*sizeof(int),cudaMemcpyHostToDevice);
    cudaMemcpy(dev_c, c , N*sizeof(int),cudaMemcpyHostToDevice);
    cudaEventCreate( &start ); // instrument code to measure start time
    cudaEventCreate( &stop );
    cudaEventRecord( start, 0 );
    vectorAdd<<<B,T>>>(dev_a,dev_b,dev_c);
    cudaMemcpy(c,dev_c,N*sizeof(int),cudaMemcpyDeviceToHost);
    cudaEventRecord( stop, 0 ); // instrument code to measure end time
    cudaEventSynchronize( stop );
    cudaEventElapsedTime( &elapsed_time_ms, start, stop );
    for(int i=0;i<N;i++) {
//        printf("%d+%d=%d\n",a[i],b[i],c[i]);
        assert(c[i]==(a[i]+b[i]));
    }
    printf("Time to calculate results: %f ms.\n", elapsed_time_ms);
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
    return 0;
}

```

Conclusion:

Thus, I have implemented a CUDA program for Vector Addition.

Name of Assessing Faculty: Prof. S.T. Shingade

Signature: _____

Date: _____

Practical 5f

Objective:

To write a CUDA program for Vector Multiplication.

Program:

```
#include <stdio.h>
#include <cuda.h>
#include <sys/time.h>
#include <assert.h>

__global__ void mult_vect(float * x, float * y, float * z, int n)
{
    int idx= blockIdx.x * blockDim.x + threadIdx.x;
    if(idx < n)
    {
        z[idx] = x[idx] * y[idx];
    }
}

int main()
{
    float *x_h, *y_h, *z_h;
    float *x_d, *y_d, *z_d;
    int n= 20,i;
    size_t size= n * sizeof(float);
    struct timeval TimeValue_Start;
    struct timezone TimeZone_Start;
    struct timeval TimeValue_Final;
    struct timezone TimeZone_Final;
    long      time_start, time_end;
    double    time_overhead;
    /* allocating memory on CPU */
    x_h= (float *)malloc(size);
    y_h= (float *)malloc(size);
```

```

z_h= (float *)malloc(size);
/* allocating memory on Device */
cudaMalloc( (void**)&x_d, size );
cudaMalloc( (void**)&y_d, size );
cudaMalloc( (void**)&z_d, size );
/* Initialization of Vectors */
for(i=0; i < n; i++)
{
    x_h[i]= (float) i;
    y_h[i]= (float) i;
}
/* Copying from Host to Device */
cudaMemcpy(x_d, x_h, size, cudaMemcpyHostToDevice);
cudaMemcpy(y_d, y_h, size, cudaMemcpyHostToDevice);
cudaMemcpy(z_d, z_h, size, cudaMemcpyHostToDevice);
int block_size= 4;
int num_blocks= (n + block_size - 1) / block_size;
gettimeofday(&TimeValue_Start, &TimeZone_Start);
/* kernel launching */
mult_vect <<<num_blocks, block_size>>> (x_d, y_d, z_d, n);
gettimeofday(&TimeValue_Final, &TimeZone_Final);
/* Copying from Device to Host */
cudaMemcpy(x_h, x_d, size, cudaMemcpyDeviceToHost);
cudaMemcpy(y_h, y_d, size, cudaMemcpyDeviceToHost);
cudaMemcpy(z_h, z_d, size, cudaMemcpyDeviceToHost);
time_end = TimeValue_Final.tv_sec * 1000000 + TimeValue_Final.tv_usec;
time_start = TimeValue_Start.tv_sec * 1000000 + TimeValue_Start.tv_usec;
time_overhead = (time_end - time_start)/1000000.0;
/* Checking whether the result is correct or not */
for(i = 0; i < n ; i++)
    assert(z_h[i]==(x_h[i] * y_h[i]));
printf("\n\t\t Time in Seconds (T)      : %lf\n\n",time_overhead);
free(x_h);
free(y_h);

```

```
    free(z_h);  
    cudaFree(x_d);  
    cudaFree(y_d);  
    cudaFree(z_d);  
return 0;  
}
```

Conclusion:

Thus, I have implemented a CUDA program for Vector Multiplication.

Name of Assessing Faculty: Prof. S.T. Shingade

Signature: _____

Date: _____