**Name:** Shivam Tiwari

**Roll No:** 5117060

**Experiment No: 1**

**Aim**: Execution of Simple Hello Word program on MPI platform.

#include "mpi.h"

#include <stdio.h>

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

{

int rank, size;

MPI\_Init( &argc, &argv );

MPI\_Comm\_rank( MPI\_COMM\_WORLD, &rank );

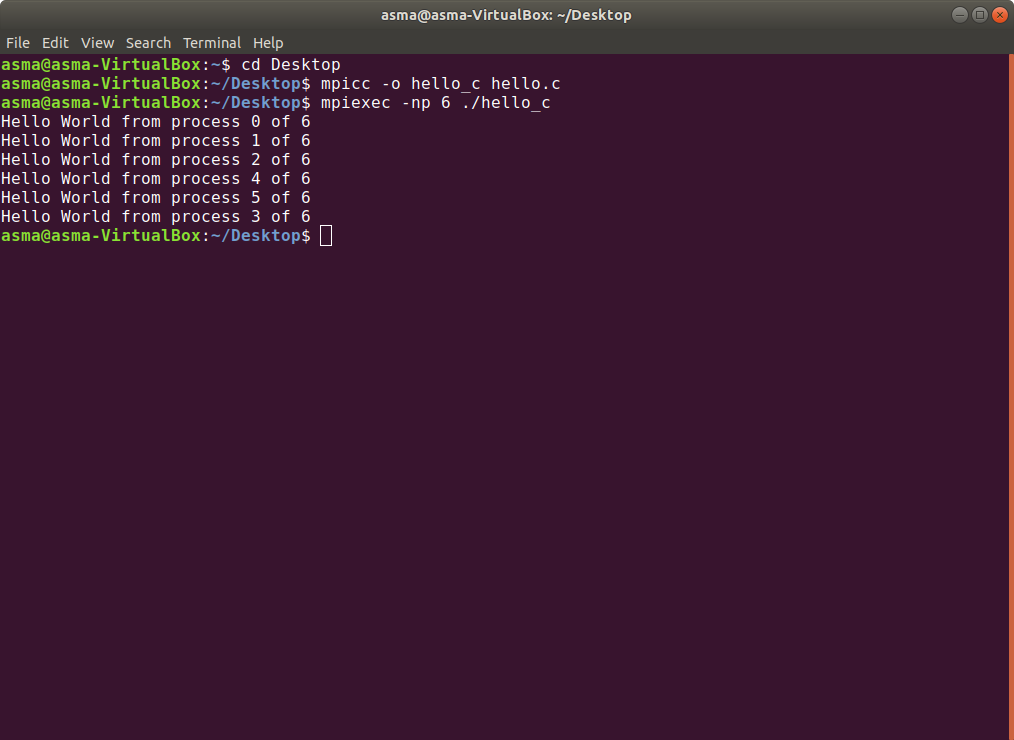
MPI\_Comm\_size( MPI\_COMM\_WORLD, &size );

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

MPI\_Finalize();

return 0;

}



**Name:** Shivam Tiwari

**Roll No:** 5117060

**Experiment No: 2**

**Aim:** a. Program to send and receive data to/from processors using MPI.

b. Program illustrating Broadcast of data using MPI.

#include "mpi.h"

#include<stdio.h>

int main(int argc, char \*\*argv)

{

MPI\_Init(NULL, NULL);

// Find out rank, size

int world\_rank;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &world\_rank);

int world\_size;

MPI\_Comm\_size(MPI\_COMM\_WORLD, &world\_size);

int number;

if (world\_rank == 0) {

number = -1;

MPI\_Send(&number, 1, MPI\_INT, 1, 0, MPI\_COMM\_WORLD);

MPI\_Send(&number, 1, MPI\_INT, 2, 0, MPI\_COMM\_WORLD);

} else if (world\_rank == 1) {

MPI\_Recv(&number, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD,

MPI\_STATUS\_IGNORE);

printf("Process 1 received number %d from process 0\n",

number);

}

if (world\_rank == 2) {

MPI\_Recv(&number, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD,

MPI\_STATUS\_IGNORE);

printf("Process 2 received number %d from process 0\n",

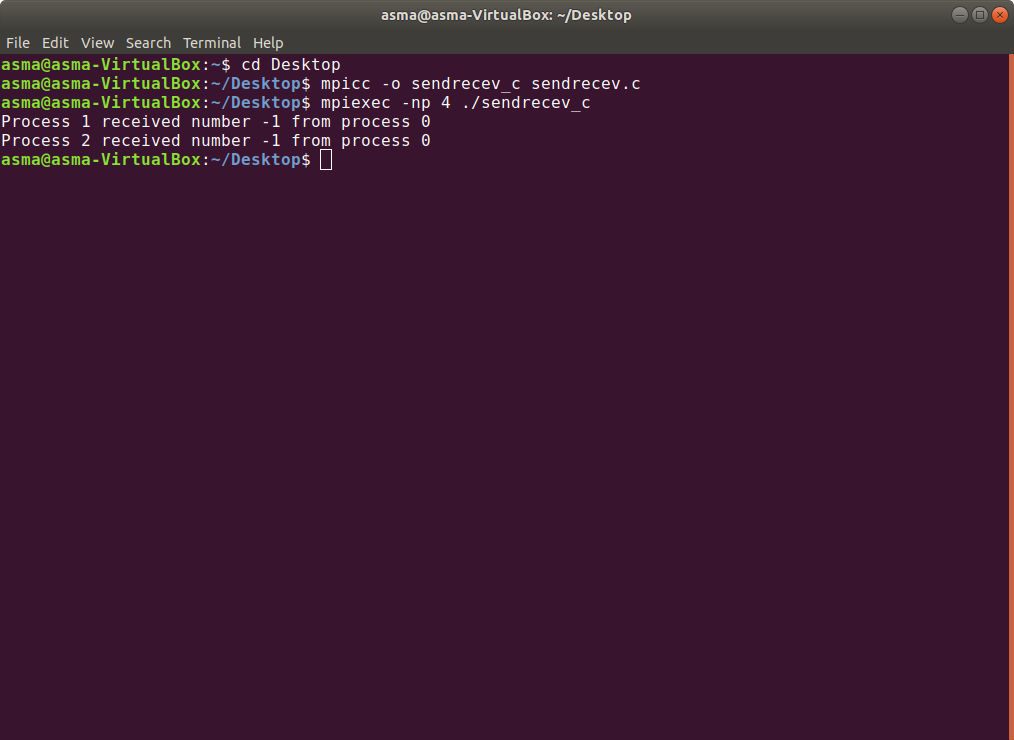
number);

}

MPI\_Finalize();

return 0;

}



**b. Program illustrating Broadcast of data using MPI.**

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

void my\_bcast(void\* data, int count, MPI\_Datatype datatype, int root,

MPI\_Comm communicator) {

int world\_rank;

MPI\_Comm\_rank(communicator, &world\_rank);

int world\_size;

MPI\_Comm\_size(communicator, &world\_size);

if (world\_rank == root) {

// If we are the root process, send our data to everyone

int i;

for (i = 0; i < world\_size; i++) {

if (i != world\_rank) {

MPI\_Send(data, count, datatype, i, 0, communicator);

}

}

} else {

// If we are a receiver process, receive the data from the root

MPI\_Recv(data, count, datatype, root, 0, communicator, MPI\_STATUS\_IGNORE);

}

}

int main(int argc, char\*\* argv) {

MPI\_Init(NULL, NULL);

int world\_rank;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &world\_rank);

int data;

if (world\_rank == 0) {

data = 100;

printf("Process 0 broadcasting data %d\n", data);

my\_bcast(&data, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

} else {

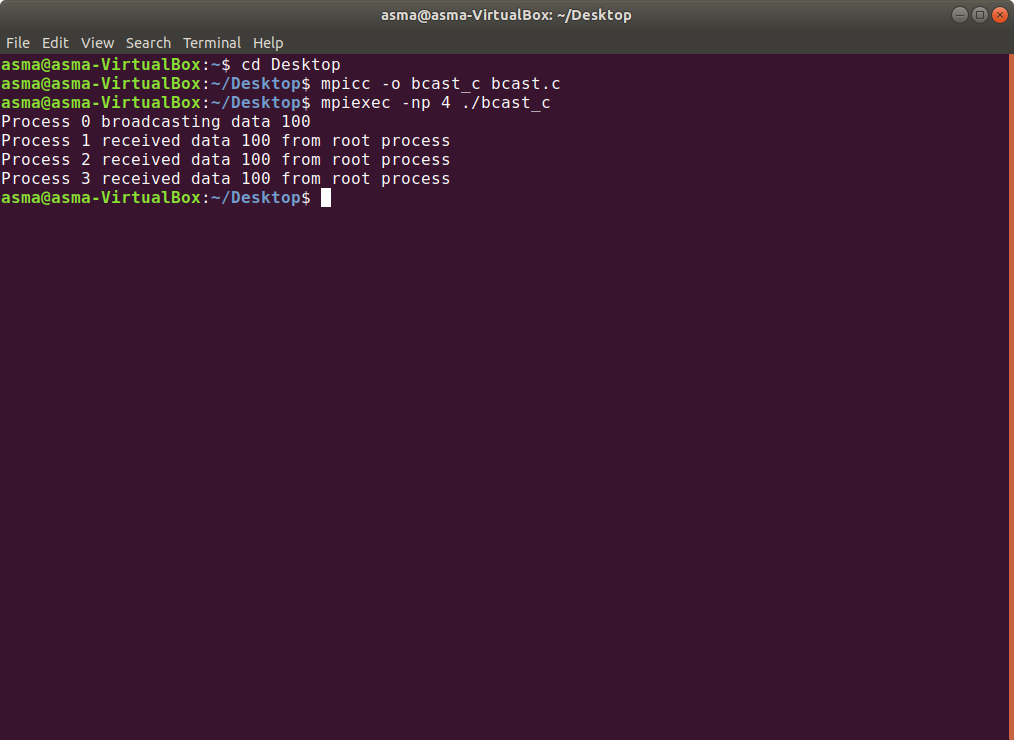
my\_bcast(&data, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

printf("Process %d received data %d from root process\n", world\_rank, data);

}

MPI\_Finalize();

}



**Name:** Shivam Tiwari

**Roll No:** 5117060

**Experiment No: 3**

**Aim:** To calculate factorial of a number.

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <mpi.h>

int main (int argc, char \*\* argv){

int rank, size,tag=100;

MPI\_Init (&argc, &argv); /\* starts MPI \*/

MPI\_Comm\_rank (MPI\_COMM\_WORLD, &rank); /\* get current process id \*/

if (rank == 0)

{

int a[10]={1,2,3,4,5,6,7,8,9,10}; // the array with the values to calculate the factorial

int fact[10] = {0}; // the array to store the results

MPI\_Send(a, 10, MPI\_INT,1,tag, MPI\_COMM\_WORLD); // the values

MPI\_Recv(fact, 10, MPI\_INT,1,tag, MPI\_COMM\_WORLD,MPI\_STATUSES\_IGNORE); // wait for the result

for(int i = 0; i < 10; i++) // print the results;

printf("Process %d,Result=%d\n",rank, fact[i]);

}

else if (rank == 1)

{

int a[10] = {0};

int fact[10] = {0};

MPI\_Recv(a, 10, MPI\_INT,0,tag, MPI\_COMM\_WORLD,MPI\_STATUSES\_IGNORE);

for(int i = 0; i < 10; i++){

int f = 1;

for (int k = 1; k <= a[i]; ++k) // Calculate the factorials

f \*= k;

fact[i] = f;

}

MPI\_Send(fact,10, MPI\_INT,0,tag, MPI\_COMM\_WORLD); // send the factorials to process 0

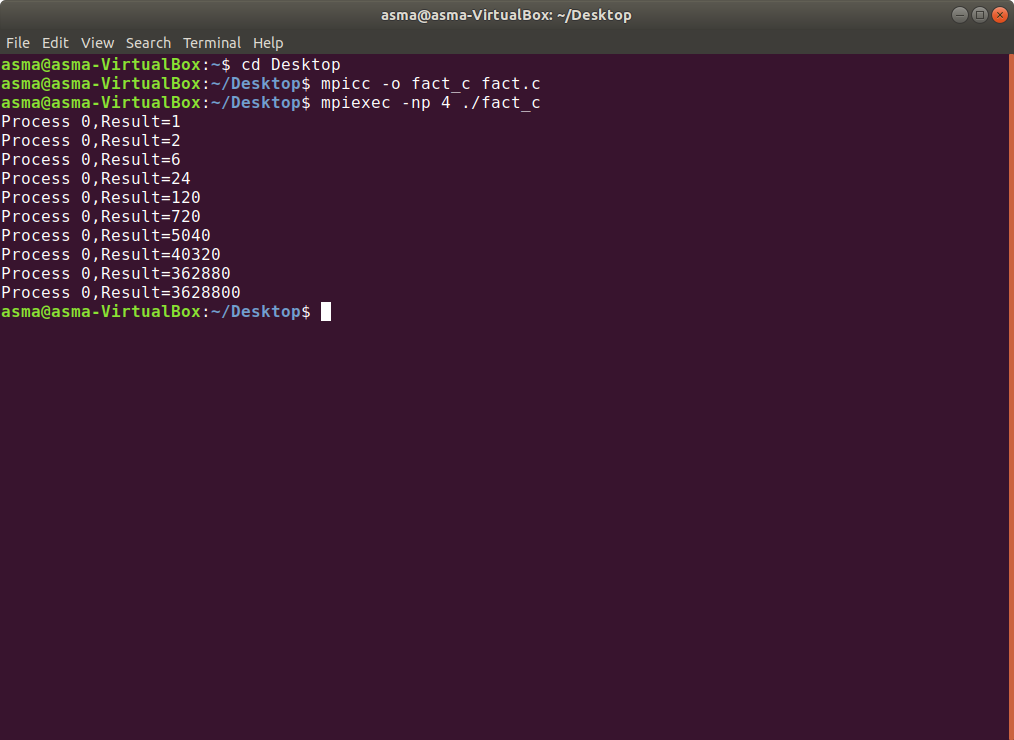
}

MPI\_Comm\_size (MPI\_COMM\_WORLD, &size); /\* get number of processes \*/

MPI\_Finalize();

return 0;

}



/\* simple parallel factorial calculator. Only useful

\* to illustrate collective communication :)

\*/

#include <stdio.h>

#include "mpi.h"

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

int myRank;

int size;

int fact;

int lower,upper;

int i;

double local\_result = 1.0;

double total;

/\* initialize MPI \*/

MPI\_Init(&argc,&argv);

/\* get my rank and the size of the communicator \*/

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myRank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

/\* get the input. (only if i have rank 0) \*/

if(myRank==0){

printf("Enter a number:");

scanf("%d",&fact);

}

/\* since only the process with rank 0 has the input,

\* we must pass it to all the other processes. \*/

MPI\_Bcast(&fact, /\* in/out parameter \*/

1, /\* count \*/

MPI\_INT, /\* datatype \*/

0, /\* root \*/

MPI\_COMM\_WORLD); /\* communicator \*/

/\* calculate the upper and lower boundaries

\* for each process

\*/

if(myRank==0){

lower = 1;

}else

lower = myRank \* (fact / size) + 1;

if(myRank==(size-1))

upper = fact;

else

upper = (myRank + 1) \* (fact / size);

/\* now that we know upper and lower, do the

\* multiplication in our local area

\*/

for(i=lower;i<=upper;i++){

local\_result = local\_result \* (double)i;

printf("\nMy upper=%d lower=%d rank=%d val=%lf",upper,lower,myRank,local\_result);

}

/\* combine all the local results by multiplying them

\* together

\*/

MPI\_Reduce(&local\_result, /\* operand \*/

&total, /\* result \*/

1, /\* count \*/

MPI\_DOUBLE, /\* datatype \*/

MPI\_PROD, /\* operator \*/

0, /\* root rank \*/

MPI\_COMM\_WORLD); /\* communicator \*/

/\* give the output to the user \*/

if(myRank==0){

printf("The factorial of %d is %lf, and was calculated using %d processes\n",fact,total,size);

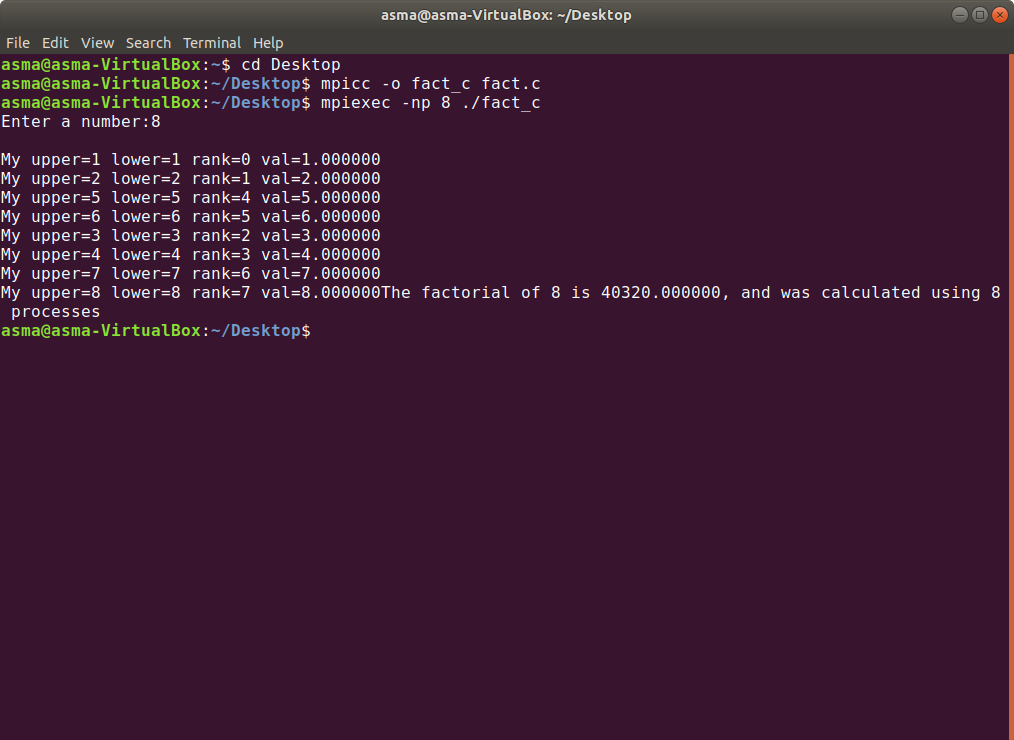
}

/\* shut down MPI \*/

MPI\_Finalize();

return 0;

}



**Name:** Shivam Tiwari

**Roll No:** 5117060

**Experiment No: 4**

**Aim:** a. To implement Average of an array.

b. To implement ring algorithm.

#include <stdio.h>

#include "mpi.h"

int main(int argc, char\*\* argv){

int my\_rank;

int total\_processes;

int root = 0;

int data[100];

int data\_loc[100];

float final\_res[100];

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &total\_processes);

int input\_size = 0;

if (my\_rank == 0){

printf("Input how many numbers: ");

scanf("%d", &input\_size);

printf("Input the elements of the array: ");

for(int i=0; i<input\_size; i++){

scanf("%d", &data[i]);

}

}

MPI\_Bcast(&input\_size, 1, MPI\_INT, root, MPI\_COMM\_WORLD);

int loc\_num = input\_size/total\_processes;

MPI\_Scatter(&data, loc\_num, MPI\_INT, data\_loc, loc\_num, MPI\_INT, root, MPI\_COMM\_WORLD);

int loc\_sum = 0;

for(int i=0; i< loc\_num; i++)

loc\_sum += data\_loc[i];

float loc\_avg = (float) loc\_sum / (float) loc\_num;

MPI\_Gather(&loc\_avg, 1, MPI\_FLOAT, final\_res, 1, MPI\_FLOAT, root, MPI\_COMM\_WORLD);

if(my\_rank==0){

float fin = 0;

for(int i=0; i<total\_processes; i++)

fin += final\_res[i];

float avg = fin / (float) total\_processes;

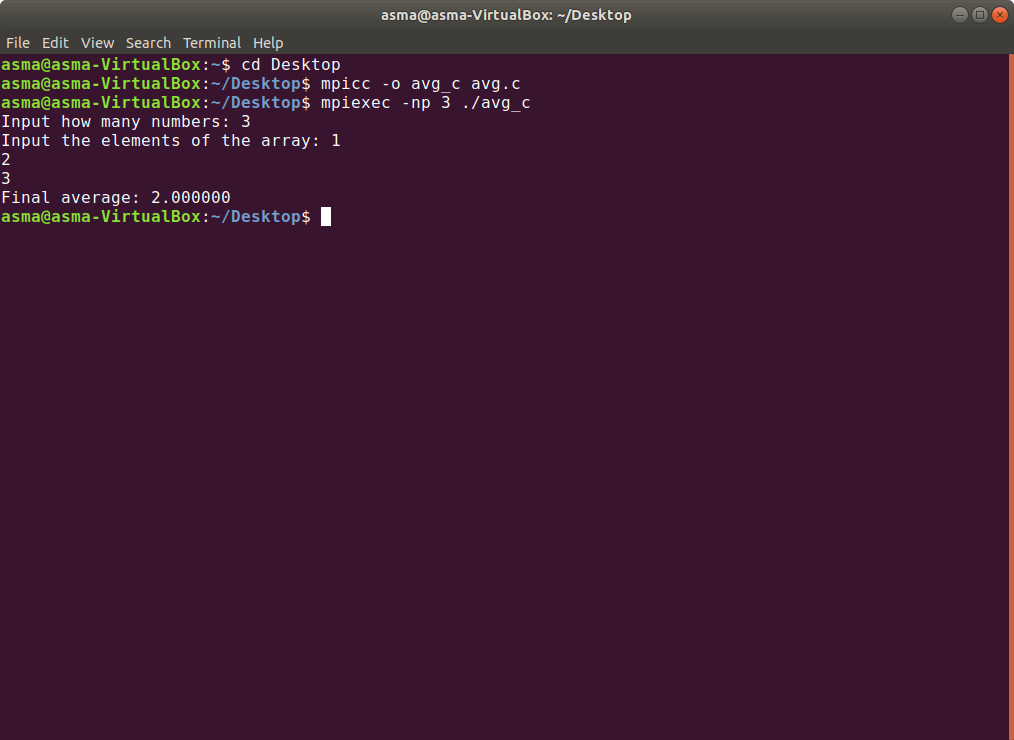
printf("Final average: %f \n", avg);

}

MPI\_Finalize();

return 0;

}



**b. To implement ring algorithm.**

#include "mpi.h"

#include<stdio.h>

int main(int argc, char \*\*argv)

{

MPI\_Init(&argc,&argv);

int world\_rank;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &world\_rank);

int world\_size;

MPI\_Comm\_size(MPI\_COMM\_WORLD, &world\_size);

int token;

if (world\_rank != 0) {

MPI\_Recv(&token, 1, MPI\_INT, world\_rank - 1, 0,

MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

printf("Process %d received token %d from process %d\n",

world\_rank, token, world\_rank - 1);

} else {

// Set the token's value if you are process 0

token = -1;

}

MPI\_Send(&token, 1, MPI\_INT, (world\_rank + 1) % world\_size,

0, MPI\_COMM\_WORLD);

// Now process 0 can receive from the last process.

if (world\_rank == 0) {

MPI\_Recv(&token, 1, MPI\_INT, world\_size - 1, 0,

MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

printf("Process %d received token %d from process %d\n",

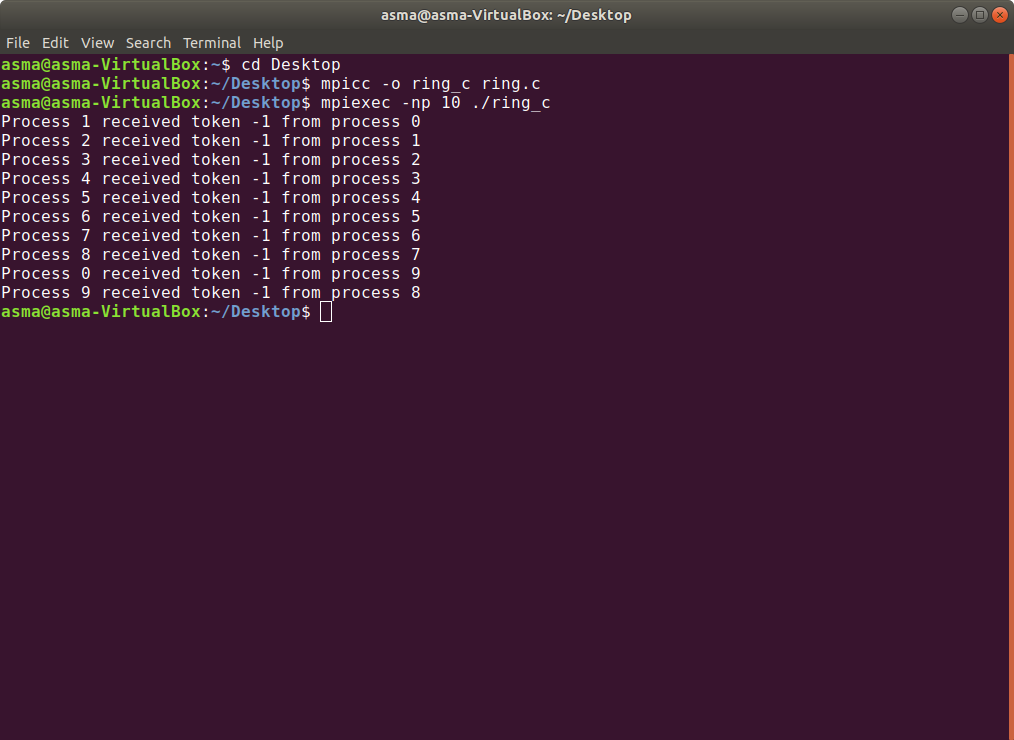
world\_rank, token, world\_size - 1);

}

MPI\_Finalize();

return 0;

}



**Name:** Shivam Tiwari

**Roll No:** 5117060

**Experiment No: 5**

**Aim:** To find sum of a one-dimensional Array.

#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);

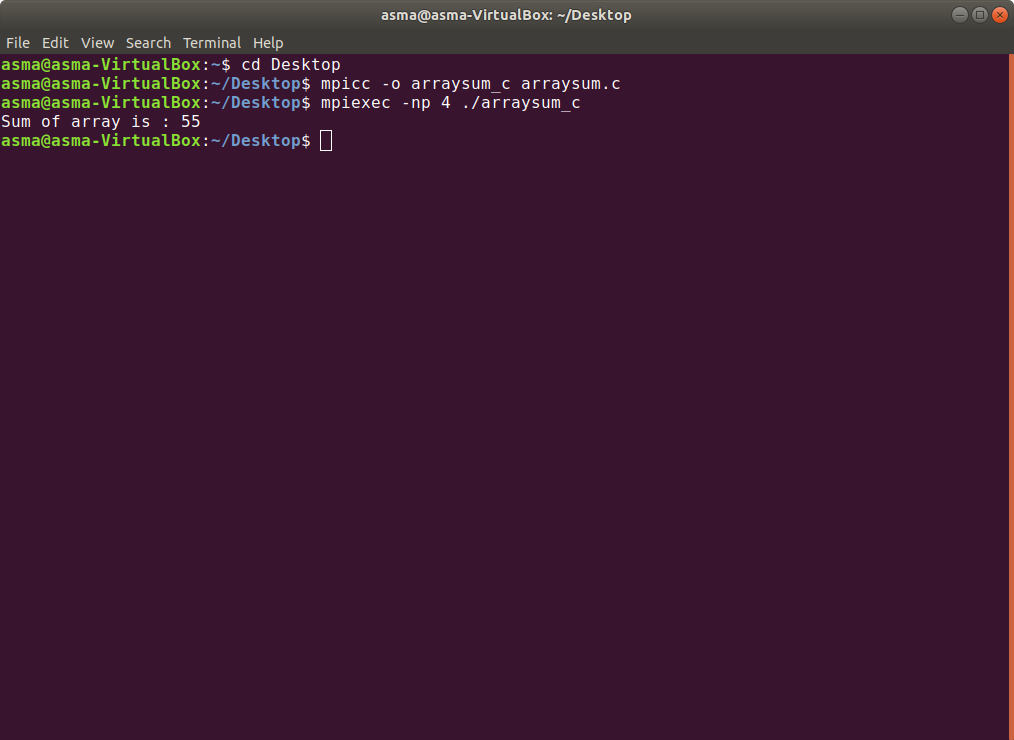
}

// cleans up all MPI state before exit of process

MPI\_Finalize();

return 0;

}

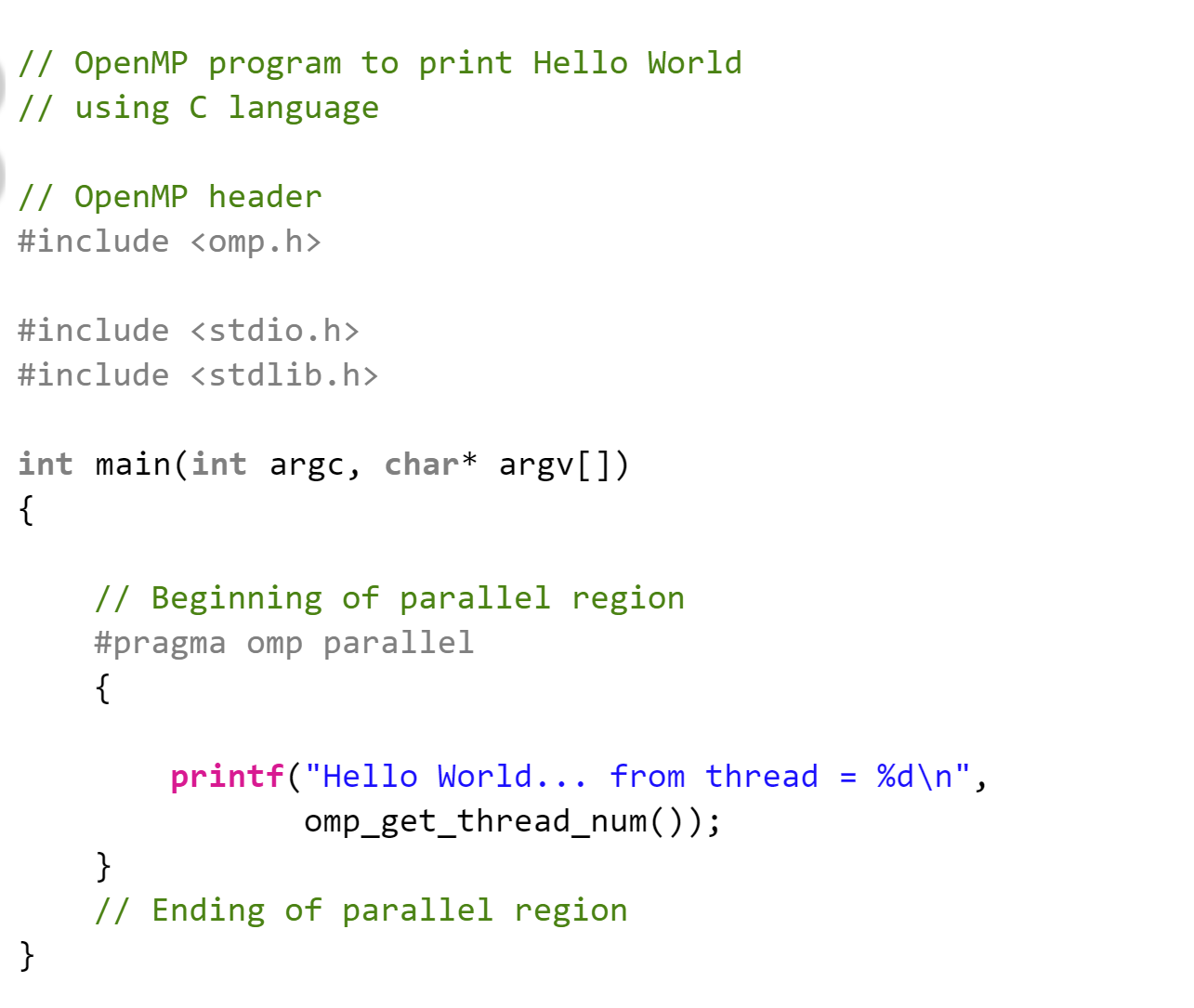


**Name:** Shivam Tiwari

**Roll No:** 5117060

**Experiment No: 6**

**Aim:** Using directives of MPI/OpenMP implement parallel programming for Hello World.



shivam@shivam -VirtualBox:~$ cd Desktop/  
shivam@shivam -VirtualBox:~/Desktop   
shivam@shivam -VirtualBox:~/Desktop$ export OMP\_NUM\_THREADS=5  
shivam@shivam -VirtualBox:~/Desktop$ gcc -o hello -fopenmp HelloworldOpenMP.c  
shivam@shivam -VirtualBox:~/Desktop $ ./hello  
Hello World... from thread = 2  
Hello World... from thread = 4  
Hello World... from thread = 1  
Hello World... from thread = 0  
Hello World... from thread = 3