**Assignment\_3**

**Submitted by: Sehajleen kaur(102203342)**

**//Pi\_MPI**

#include <stdio.h>

#include <mpi.h>

#include <stdlib.h>

#define NUM\_STEPS 10000

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

int size, rank;

double step, local\_sum=0.0, x, partial\_pi, pi;

MPI\_Init(&argc, &argv);

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

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

long num\_steps=NUM\_STEPS;

step=1.0/(double)num\_steps;

MPI\_Bcast(

&num\_steps,

1,

MPI\_LONG,

0,

MPI\_COMM\_WORLD

);

for(int i=rank; i<num\_steps; i+=size){

x=(i+0.5)\*step;

local\_sum+=4.0/(1.0+x\*x);

}

partial\_pi=step\*local\_sum;

MPI\_Reduce(

&partial\_pi,

&pi,

1,

MPI\_DOUBLE,

MPI\_SUM,

0,

MPI\_COMM\_WORLD

);

if(rank==0){

printf("Approximaiton of Pi: %f\n", pi);

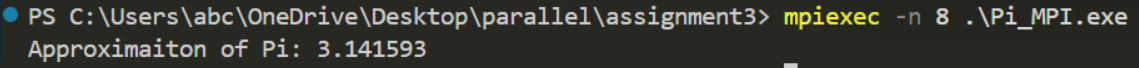
}

MPI\_Finalize();

return 0;

}

**//Output:**



**// DAXPY Loop**

#include <stdio.h>

#include <mpi.h>

#include <stdlib.h>

#define N (1 << 16) // Vector size (2^16)

#define A 2.5 // Scalar value

void daxpy\_serial(double \*x, double \*y, double a, int n){

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

x[i]=a\*x[i]+y[i];

}

}

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

int rank, size;

double \*X, \*Y, \*local\_X, \*local\_Y;

int chunk\_size;

double start, end, serial\_time, parallel\_time;

MPI\_Init(&argc, &argv);

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

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

chunk\_size=N/size;

local\_X=(double\*)malloc(chunk\_size\*sizeof(double));

local\_Y=(double\*)malloc(chunk\_size\*sizeof(double));

if(rank==0){

X=(double\*)malloc(N\*sizeof(double));

Y=(double\*)malloc(N\*sizeof(double));

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

X[i]=1.0;

Y[i]=2.0;

}

start=MPI\_Wtime();

daxpy\_serial(X, Y, A, N);

end=MPI\_Wtime();

serial\_time=end-start;

printf("serial execution time: %f seconds\n", serial\_time);

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

X[i]=1.0;

Y[i]=2.0;

}

}

MPI\_Scatter(

X,

chunk\_size,

MPI\_DOUBLE,

local\_X,

chunk\_size,

MPI\_DOUBLE,

0,

MPI\_COMM\_WORLD

);

MPI\_Scatter(

Y,

chunk\_size,

MPI\_DOUBLE,

local\_Y,

chunk\_size,

MPI\_DOUBLE,

0,

MPI\_COMM\_WORLD

);

start=MPI\_Wtime();

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

local\_X[i]=A\*local\_X[i]+local\_Y[i];

}

end=MPI\_Wtime();

parallel\_time=end-start;

MPI\_Gather(

local\_X,

chunk\_size,

MPI\_DOUBLE,

X,

chunk\_size,

MPI\_DOUBLE,

0,

MPI\_COMM\_WORLD

);

if(rank==0){

printf("Parallel execution time: %f seconds\n", parallel\_time);

printf("Speedup: %f\n", serial\_time/parallel\_time);

free(X);

free(Y);

}

free(local\_X);

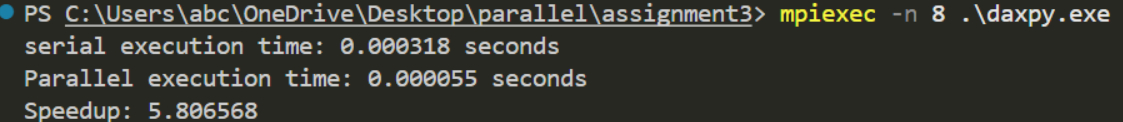
free(local\_Y);

MPI\_Finalize();

return 0;

}

**//Output:**



**//Positive\_Prime**

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

#include <math.h>

#define MAX\_VALUE 100 // Maximum number to check for primes

#define MASTER 0

int is\_prime(int num)

{

if (num < 2)

return 0;

for (int i = 2; i <= sqrt(num); i++)

{

if (num % i == 0)

return 0;

}

return 1;

}

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);

if (rank == MASTER)

{

int candidate = 2;

int workers = size - 1;

MPI\_Status status;

int received, source;

// Send initial tasks to workers

for (int i = 1; i <= workers && candidate <= MAX\_VALUE; i++)

{

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

candidate++;

}

// Receive results and assign new tasks

while (candidate <= MAX\_VALUE)

{

MPI\_Recv(&received, 1, MPI\_INT, MPI\_ANY\_SOURCE, 0, MPI\_COMM\_WORLD, &status);

source = status.MPI\_SOURCE;

if (received > 0)

{

printf("Prime: %d\n", received);

}

// Send a new number to check

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

candidate++;

}

// Collect remaining results from workers

for (int i = 1; i <= workers; i++)

{

MPI\_Recv(&received, 1, MPI\_INT, MPI\_ANY\_SOURCE, 0, MPI\_COMM\_WORLD, &status);

if (received > 0)

{

printf("Prime: %d\n", received);

}

}

// Send termination signal to all workers

int stop\_signal = -1;

for (int i = 1; i <= workers; i++)

{

MPI\_Send(&stop\_signal, 1, MPI\_INT, i, 0, MPI\_COMM\_WORLD);

}

}

else

{

// Worker processes

while (1)

{

int test\_number;

MPI\_Recv(&test\_number, 1, MPI\_INT, MASTER, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

if (test\_number == -1)

break; // Exit condition

int result = is\_prime(test\_number) ? test\_number : -1;

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

}

}

MPI\_Finalize();

return 0;

}

**//Output:**

