**Assignment\_2**

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

**//Dot\_Product**

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

#define N 8 //size of vector

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

int rank, num\_procs;

int local\_N;

int A[N]={1,2,3,4,5,6,7,8};

int B[N]={8,7,6,5,4,3,2,1};

MPI\_Init(&argc, &argv);

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

MPI\_Comm\_size(MPI\_COMM\_WORLD, &num\_procs);

local\_N=N/num\_procs; //workload divide

int local\_A[local\_N], local\_B[local\_N], local\_dot\_product=0, global\_dot\_product=0;

//scattering data

MPI\_Scatter(

A,

local\_N,

MPI\_INT,

local\_A,

local\_N,

MPI\_INT,

0,

MPI\_COMM\_WORLD

);

MPI\_Scatter(

B,

local\_N,

MPI\_INT,

local\_B,

local\_N,

MPI\_INT,

0,

MPI\_COMM\_WORLD

);

//compute dot product

for(int i=0; i<local\_N; i++){

local\_dot\_product+=local\_A[i]\*local\_B[i];

}

//reduce

MPI\_Reduce(

&local\_dot\_product,

&global\_dot\_product,

1,

MPI\_INT,

MPI\_SUM,

0,

MPI\_COMM\_WORLD

);

if (rank == 0) {

printf("Dot Product: %d\n", global\_dot\_product);

}

MPI\_Finalize();

return 0;

}

**//Output:**



**//heatDistributionSimulation**

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

#include <math.h>

#define N 10 // grid size

#define MAX\_ITER 1000

#define THRESHOLD 0.01

void initialize\_grid(double grid[N][N])

{

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

{

for (int j = 0; j < N; j++)

{

if (i == 0 || i == N - 1 || j == 0 || j == N - 1)

grid[i][j] = 100.0; // boundary conditions

else

grid[i][j] = 0.0; // interior points

}

}

}

void print\_grid(double grid[N][N])

{

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

{

for (int j = 0; j < N; j++)

{

printf("%.1f ", grid[i][j]);

}

printf("\n");

}

printf("\n");

}

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

{

int rank, num\_procs, rows\_per\_proc;

MPI\_Init(&argc, &argv);

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

MPI\_Comm\_size(MPI\_COMM\_WORLD, &num\_procs);

rows\_per\_proc = N / num\_procs;

if (N % num\_procs != 0)

{

if (rank == 0)

printf("Error: grid rows must be divisble by number of processes.\n");

MPI\_Finalize();

return -1;

}

double grid[N][N], local\_grid[rows\_per\_proc + 2][N], new\_grid[rows\_per\_proc + 2][N];

if (rank == 0)

initialize\_grid(grid);

// scatter the grid to processes

MPI\_Scatter(

grid,

rows\_per\_proc \* N,

MPI\_DOUBLE,

&local\_grid[1][0],

rows\_per\_proc \* N,

MPI\_DOUBLE,

0,

MPI\_COMM\_WORLD);

// Main computation loop

for (int iter = 0; iter < MAX\_ITER; iter++)

{

if (rank > 0)

{

MPI\_Send(

&local\_grid[1][0],

N,

MPI\_DOUBLE,

rank - 1,

0,

MPI\_COMM\_WORLD);

MPI\_Recv(

&local\_grid[0][0],

N,

MPI\_DOUBLE,

rank - 1,

0,

MPI\_COMM\_WORLD,

MPI\_STATUS\_IGNORE);

}

if (rank < num\_procs - 1)

{

MPI\_Send(

&local\_grid[rows\_per\_proc][0],

N,

MPI\_DOUBLE,

rank + 1,

0,

MPI\_COMM\_WORLD);

MPI\_Recv(

&local\_grid[rows\_per\_proc + 1][0],

N,

MPI\_DOUBLE,

rank + 1,

0,

MPI\_COMM\_WORLD,

MPI\_STATUS\_IGNORE

);

}

// computing new temp values

double max\_diff = 0.0;

for (int i = 1; i <= rows\_per\_proc; i++)

{

for (int j = 1; j < N - 1; j++)

{

new\_grid[i][j] = 0.25 \* (local\_grid[i - 1][j] + local\_grid[i + 1][j] + local\_grid[i][j - 1] + local\_grid[i][j + 1]);

double diff = fabs(new\_grid[i][j] - local\_grid[i][j]);

if (diff > max\_diff)

max\_diff = diff;

}

}

//swap grids

for(int i=1; i<=rows\_per\_proc; i++){

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

local\_grid[i][j]=new\_grid[i][j];

}

}

double global\_max\_diff;

MPI\_Allreduce(

&max\_diff,

&global\_max\_diff,

1,

MPI\_DOUBLE,

MPI\_MAX,

MPI\_COMM\_WORLD);

if (global\_max\_diff < THRESHOLD) break;

}

MPI\_Gather(

&local\_grid[1][0],

rows\_per\_proc\*N,

MPI\_DOUBLE,

grid,

rows\_per\_proc\*N,

MPI\_DOUBLE,

0,

MPI\_COMM\_WORLD

);

if(rank==0){

printf("Final Heat Distribution.\n");

print\_grid(grid);

}

MPI\_Finalize();

return 0;

}

**//Output:**

A screenshot of a computer program

AI-generated content may be incorrect.

**//Matrix Multiplication**

#include <stdio.h>

#include <mpi.h>

#include <stdlib.h>

#define N 70

void multiply\_matrices(double A[N][N], double B[N][N], double C[N][N], int start, int end){

for(int i=start; i<end; i++){

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

C[i][j] = 0;

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

C[i][j] += A[i][k] \* B[k][j];

}

}

}

}

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

int rank, size;

double A[N][N], B[N][N], C[N][N];

double local\_A[N][N], local\_C[N][N];

double start\_time, run\_time;

MPI\_Init(&argc, &argv);

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

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

int rows\_per\_process=N/size;

int start=rank\*rows\_per\_process;

int end=(rank == size - 1) ? N : start + rows\_per\_process;

if(rank==0){

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

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

A[i][j]=rand()%10;

B[i][j]=rand()%10;

}

}

start\_time=MPI\_Wtime();

}

MPI\_Bcast(

B,

N\*N,

MPI\_DOUBLE,

0,

MPI\_COMM\_WORLD

);

MPI\_Scatter(

A,

rows\_per\_process\*N,

MPI\_DOUBLE,

local\_A,

rows\_per\_process\*N,

MPI\_DOUBLE,

0,

MPI\_COMM\_WORLD

);

multiply\_matrices(local\_A, B, local\_C, start, end);

MPI\_Gather(

local\_C,

rows\_per\_process\*N,

MPI\_DOUBLE,

C,

rows\_per\_process\*N,

MPI\_DOUBLE,

0,

MPI\_COMM\_WORLD

);

if (rank == 0) {

run\_time = MPI\_Wtime() - start\_time;

printf("MPI Matrix Multiplication Time: %f seconds\n", run\_time);

}

MPI\_Finalize();

return 0;

}

**//Output:**



**//Matrix Transpose**

#include <stdio.h>

#include <mpi.h>

#include <stdlib.h>

#define N 4

void print\_matrix(int matrix[N][N], const char\* label){

printf("%s:\n", label);

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

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

printf("%d ", matrix[i][j]);

}

printf("\n");

}

printf("\n");

}

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

int rank, size;

int matrix[N][N], transpose[N][N];

int local\_row[N];

MPI\_Init(&argc, &argv);

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

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

if(size!=N){

if(rank==0){

printf("this program requires exactly %d no. of processes\n", N);

}

MPI\_Finalize();

}

if(rank==0){

int count=1;

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

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

matrix[i][j] = count++;

}

}

print\_matrix(matrix, "Original Matrix");

}

MPI\_Scatter(

matrix,

N,

MPI\_INT,

local\_row,

N,

MPI\_INT,

0,

MPI\_COMM\_WORLD

);

int local\_col[N];

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

local\_col[i]=local\_row[i];

}

MPI\_Gather(

local\_col,

N,

MPI\_INT,

transpose,

N,

MPI\_INT,

0,

MPI\_COMM\_WORLD

);

if(rank==0){

print\_matrix(transpose, "Transposed Matrix");

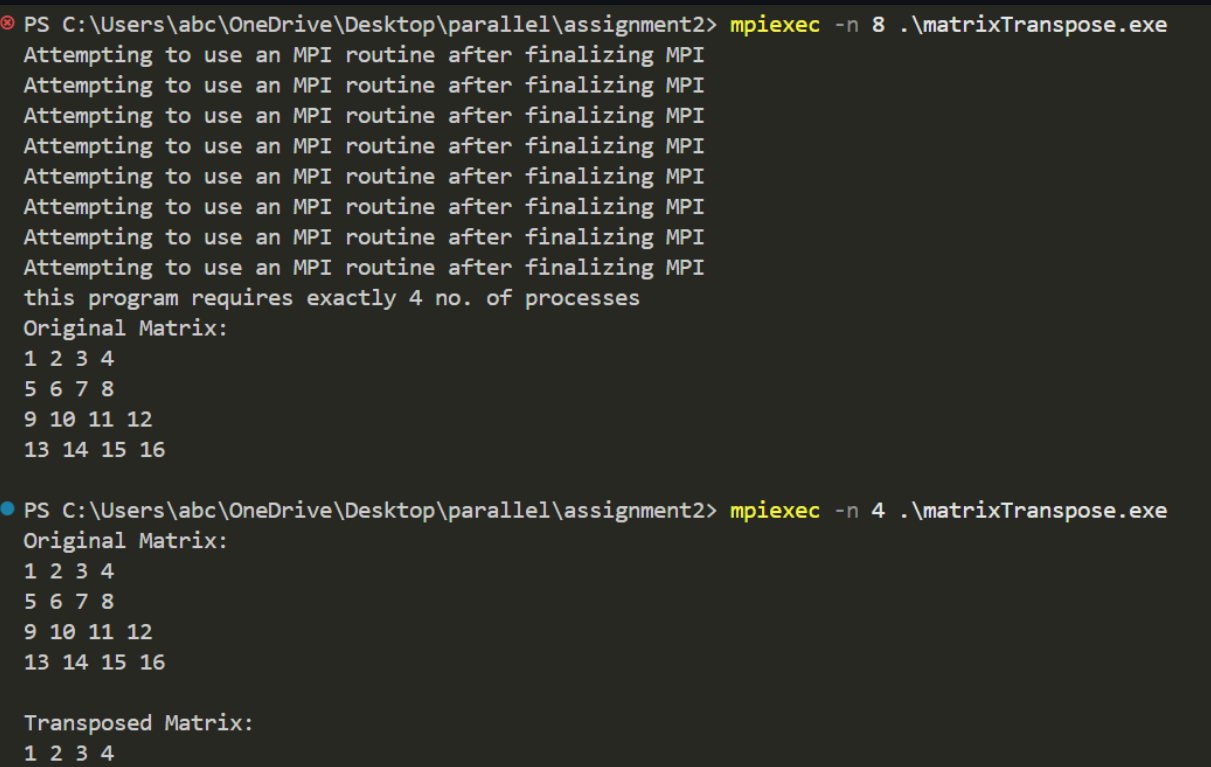
}

MPI\_Finalize();

return 0;

}

**//Output:**





**//Odd\_Even**

#include <stdio.h>

#include <mpi.h>

#include <stdlib.h>

void even\_odd\_sort(int \*local\_array, int local\_size, int global\_size, int rank, int num\_procs){

int phase, partner, temp;

MPI\_Status status;

for(phase=0; phase<global\_size; phase++){

if(phase%2==0){ //even phase

if(rank%2==0)

partner=rank+1;

else

partner=rank-1;

}

else{ //odd phase

if(rank%2==0)

partner=rank-1;

else

partner=rank+1;

}

if(partner>=0 && partner<num\_procs){

int received;

MPI\_Sendrecv(

&local\_array[0],

1,

MPI\_INT,

partner,

0,

&received,

1,

MPI\_INT,

partner,

0,

MPI\_COMM\_WORLD,

&status

);

if((rank<partner && local\_array[0] > received) || (rank>partner && local\_array[0]<received)){

temp=local\_array[0];

local\_array[0]=received;

received=temp;

}

MPI\_Sendrecv(

&received,

1,

MPI\_INT,

partner,

0,

&local\_array[0],

1,

MPI\_INT,

partner,

0,

MPI\_COMM\_WORLD,

&status

);

}

}

}

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

int rank, num\_procs;

int global\_size=8;

int local\_array[1];

MPI\_Init(&argc, &argv);

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

MPI\_Comm\_size(MPI\_COMM\_WORLD, &num\_procs);

if(num\_procs!=global\_size){

if(rank==0){

printf("Error: the number of processes must be equal to the number of elemnts.\n");

}

MPI\_Finalize();

return 1;

}

int global\_array[]={9,7,3,5,1,8,6,2};

MPI\_Scatter(

global\_array,

1,

MPI\_INT,

local\_array,

1,

MPI\_INT,

0,

MPI\_COMM\_WORLD

);

if(rank==1){

printf("Inital Array: ");

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

printf("%d ", global\_array[i]);

}

printf("\n");

}

even\_odd\_sort(

local\_array,

1,

global\_size,

rank,

num\_procs

);

MPI\_Gather(

local\_array,

1,

MPI\_INT,

global\_array,

1,

MPI\_INT,

0,

MPI\_COMM\_WORLD

);

if(rank==0){

printf("sorted array: ");

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

printf("%d ", global\_array[i]);

}

printf("\n");

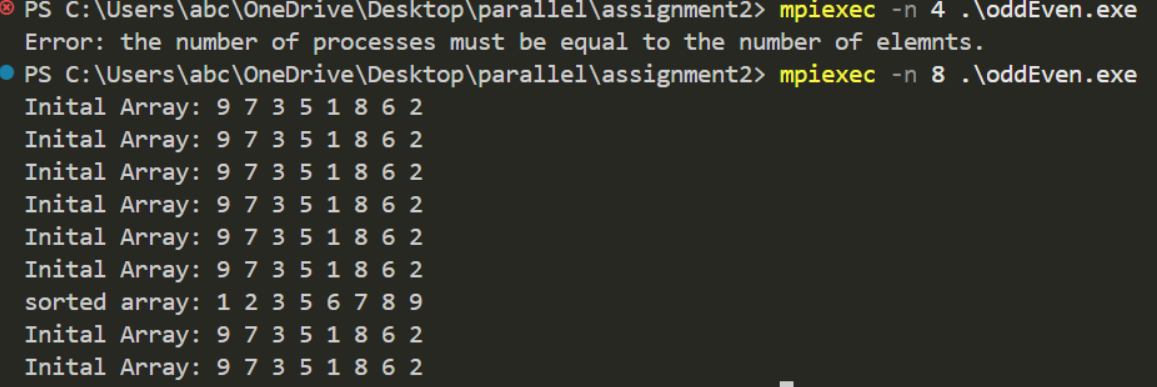
}

MPI\_Finalize();

return 0;

}

**//Output:**



**//Parallel\_Reduction**

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

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

int rank, num\_procs;

int global\_sum, local\_value;

MPI\_Init(&argc, &argv);

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

MPI\_Comm\_size(MPI\_COMM\_WORLD, &num\_procs);

local\_value=rank+1;

MPI\_Reduce(

&local\_value,

&global\_sum,

1,

MPI\_INT,

MPI\_SUM,

0,

MPI\_COMM\_WORLD

);

if(rank==0){

printf("total sum using MPI\_Reduce: %d\n", global\_sum);

}

MPI\_Finalize();

return 0;

}

**//Value of Pi**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <mpi.h>

#define NO\_OF\_ITERATIONS 1000000

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

{

int rank, size;

long long int circle\_points = 0, square\_points = 0, interval = 0;

double x, y, d;

double pi;

MPI\_Init(&argc, &argv);

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

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

srand(time(NULL) + rank);

while (interval < NO\_OF\_ITERATIONS / size)

{

x = (double)rand() / RAND\_MAX;

y = (double)rand() / RAND\_MAX;

d = x \* x + y \* y;

if (d <= 1)

{

circle\_points++;

}

square\_points++;

interval++;

}

long long int global\_circle\_points, global\_square\_points;

MPI\_Reduce(&circle\_points, &global\_circle\_points, 1, MPI\_LONG\_LONG\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);

MPI\_Reduce(&square\_points, &global\_square\_points, 1, MPI\_LONG\_LONG\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);

if (rank == 0)

{

pi = 4.0 \* (double)global\_circle\_points / (double)global\_square\_points;

printf("%f", pi);

}

return 0;

}

**//PrefixSum**

#include <stdio.h>

#include <mpi.h>

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

int rank, size;

int local\_value, prefix\_sum;

MPI\_Init(&argc, &argv);

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

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

local\_value=rank+1;

MPI\_Scan(

&local\_value,

&prefix\_sum,

1,

MPI\_INT,

MPI\_SUM,

MPI\_COMM\_WORLD

);

printf("Process %d: Local value = %d, Prefix sum = %d\n", rank, local\_value, prefix\_sum);

MPI\_Finalize();

return 0;

}