# 

**NETAJI SUBHAS UNIVERSITY OF TECHNOLOGY**

**High Performance Computing**

**[COCSC18]**

**Lab File**

SUBMITTED BY :-

**Shivam Singla**

**2019UCO1526**

**COE-1**

**Index**

|  |  |  |
| --- | --- | --- |
| S. No. | Topic | Page No. |
| 1. | Run a basic hello world program using pthreads | 3-4 |
| 2. | Run a program to find the sum of all elements of  an array using 2 processors | 5-6 |
| 3. | Compute the sum of all the elements of an array  using p processors | 7-8 |
| 4. | Write a program to illustrate basic MPI  communication routines | 9-10 |
| 5. | Design a parallel program for summing up an  array, matrix multiplication and show logging  and tracing MPI activity | 11-14 |
| 6. | Write a C program with openMP to implement  loop work sharing | 15-17 |
| 7. | Write a C program with openMP to implement  sections work sharing | 18-19 |
| 8. | Write a program to illustrate process  synchronization and collective data movements | 20-22 |

Experiment 1

Run a basic hello world program using pthreads

Code:

#include <mpi.h>

#include <stdio.h>

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

MPI\_Init(NULL, NULL);

int world\_size;

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

int world\_rank;

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

char processor\_name[MPI\_MAX\_PROCESSOR\_NAME];

int name\_len;

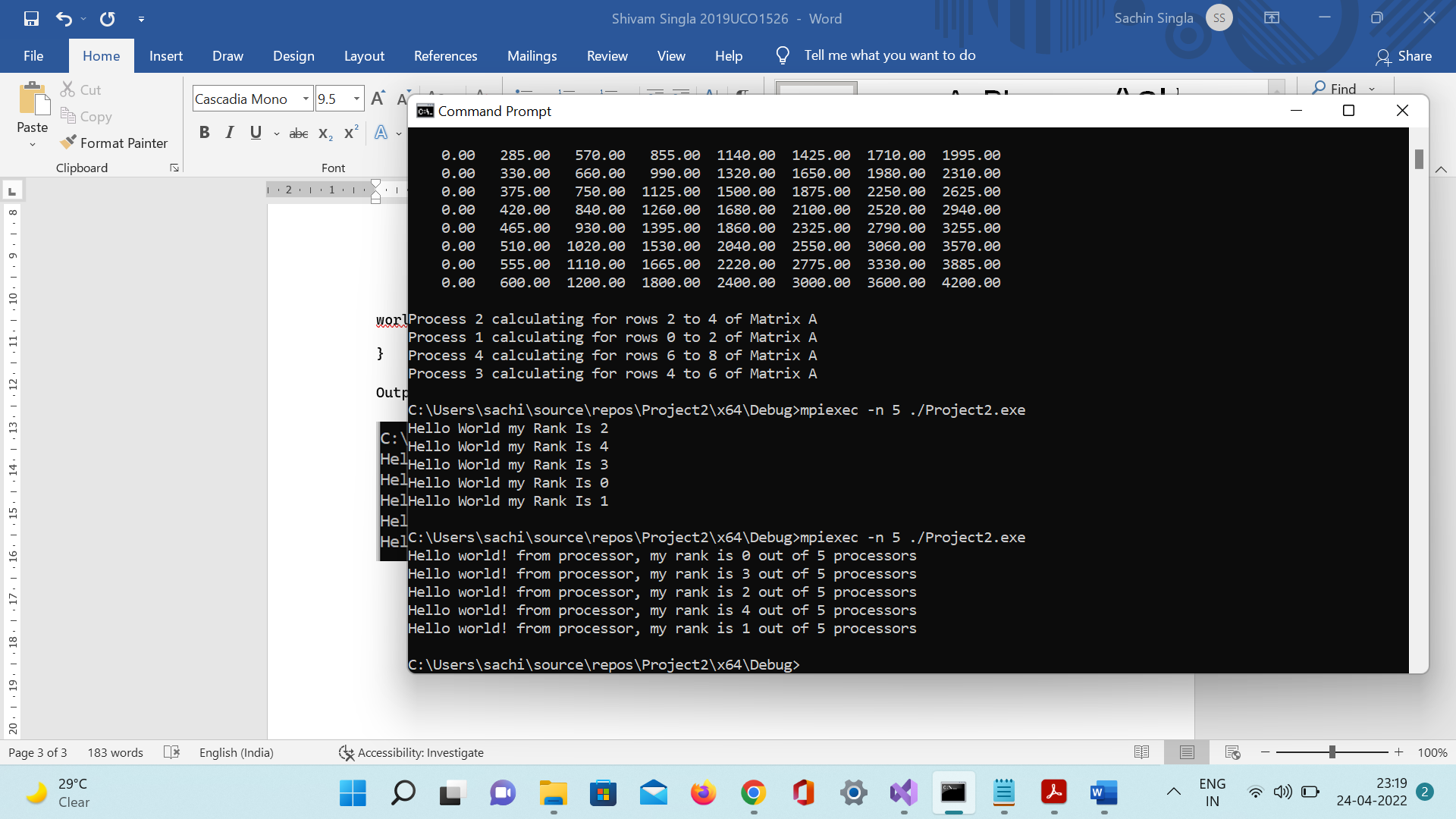
MPI\_Get\_processor\_name(processor\_name, &name\_len);

printf("Hello world! from processor, my rank is %d out of %d processors\n", world\_rank, world\_size);

MPI\_Finalize();

}

Output:



Using Pthreads

Code:

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

int thread\_count;

void\* Hello(void\* rank);

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

{

long thread;

pthread\_t\* thread\_handles;

thread\_count = strtol(argv[1], NULL, 10);

thread\_handles = malloc(thread\_count \* sizeof(pthread\_t));

for (thread = 0; thread < thread\_count; thread++)

pthread\_create(&thread\_handles[thread], NULL, Hello, (void\*)thread);

printf("Hello from the main thread\n");

for (thread = 0; thread < thread\_count; thread++)

pthread\_join(thread\_handles[thread], NULL);

free(thread\_handles);

return 0;

}

void\* Hello(void\* rank)

{

long my\_rank = (long)rank;

printf("Hello from thread %ld of %d\n", my\_rank, thread\_count);

return NULL;

}

Output:



Experiment 2

Run a program to find the sum of all elements of an array using 2 processors

Code:

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

#define n 9

int a[] = { 1, 5, 2, 6, 2, 0, 1, 9 };

int a2[1000];

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

{

int pid, np,

elements\_per\_process,

n\_elements\_recieved;

MPI\_Status status;

MPI\_Init(&argc, &argv);

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

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

if (pid == 0) {

int index, i;

elements\_per\_process = n / np;

if (np > 1) {

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);

}

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);

}

int sum = 0;

for (i = 0; i < elements\_per\_process; i++)

sum += a[i];

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;

}

printf("Sum of array is : %d\n", sum);

}

else {

MPI\_Recv(&n\_elements\_recieved,

1, MPI\_INT, 0, 0,

MPI\_COMM\_WORLD,

&status);

MPI\_Recv(&a2, n\_elements\_recieved,

MPI\_INT, 0, 0,

MPI\_COMM\_WORLD,

&status);

int partial\_sum = 0;

for (int i = 0; i < n\_elements\_recieved; i++)

partial\_sum += a2[i];

MPI\_Send(&partial\_sum, 1, MPI\_INT,

0, 0, MPI\_COMM\_WORLD);

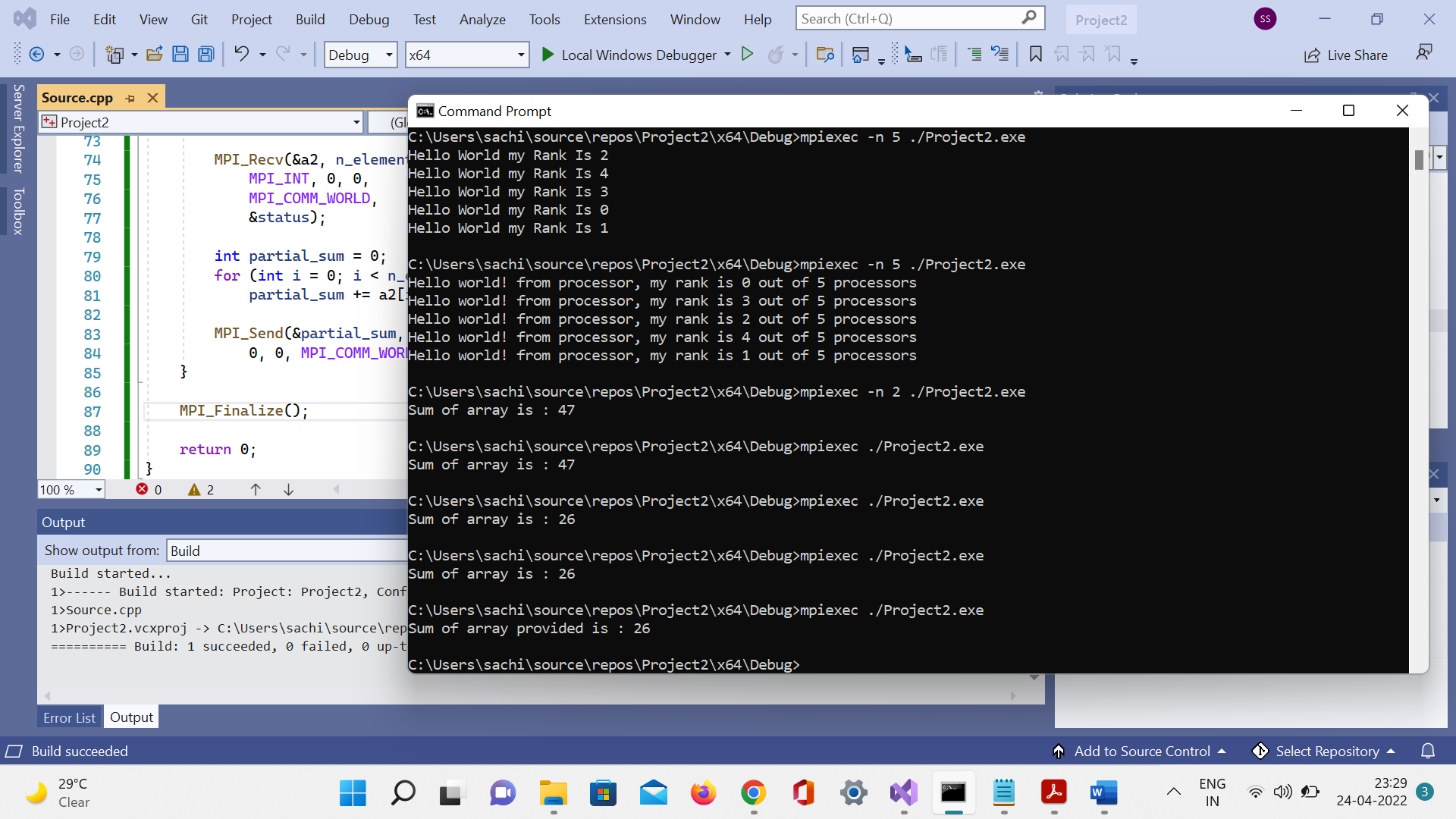
}

MPI\_Finalize();

return 0;

}

Output:



Experiment 3

Compute the sum of all the elements of an array using p processors

Code:

#include "mpi.h"

#include <stdio.h>

#include <stdlib.h>

#define n 8

int a[] = { 1, 5, 2, 6, 2, 0, 1, 9 };

// Temporary array for other processes

int b[1000];

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

{

int process\_id, no\_of\_process,

elements\_per\_process,

n\_elements\_recieved;

MPI\_Status status;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &process\_id);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &no\_of\_process);

// For process 0

if (process\_id == 0) {

int index, i;

elements\_per\_process = n / no\_of\_process;

if (no\_of\_process > 1) {

for (i = 1; i < no\_of\_process - 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);

}

// sum by process 0

int sum = 0;

for (i = 0; i < elements\_per\_process; i++)

sum += a[i];

printf("Sum by this Slave Process is %d = %d\n", process\_id, sum);

// partial sums from other processes

int tmp;

for (i = 1; i < no\_of\_process; 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("Final Sum of array by the Master Process is : %d\n", sum);

}

// Other processes

else {

MPI\_Recv(&n\_elements\_recieved, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, &status);

MPI\_Recv(&b, n\_elements\_recieved, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, &status);

int partial\_sum = 0;

for (int i = 0; i < n\_elements\_recieved; i++)

partial\_sum += b[i];

printf("Sum by process %d = %d\n", process\_id, partial\_sum);

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

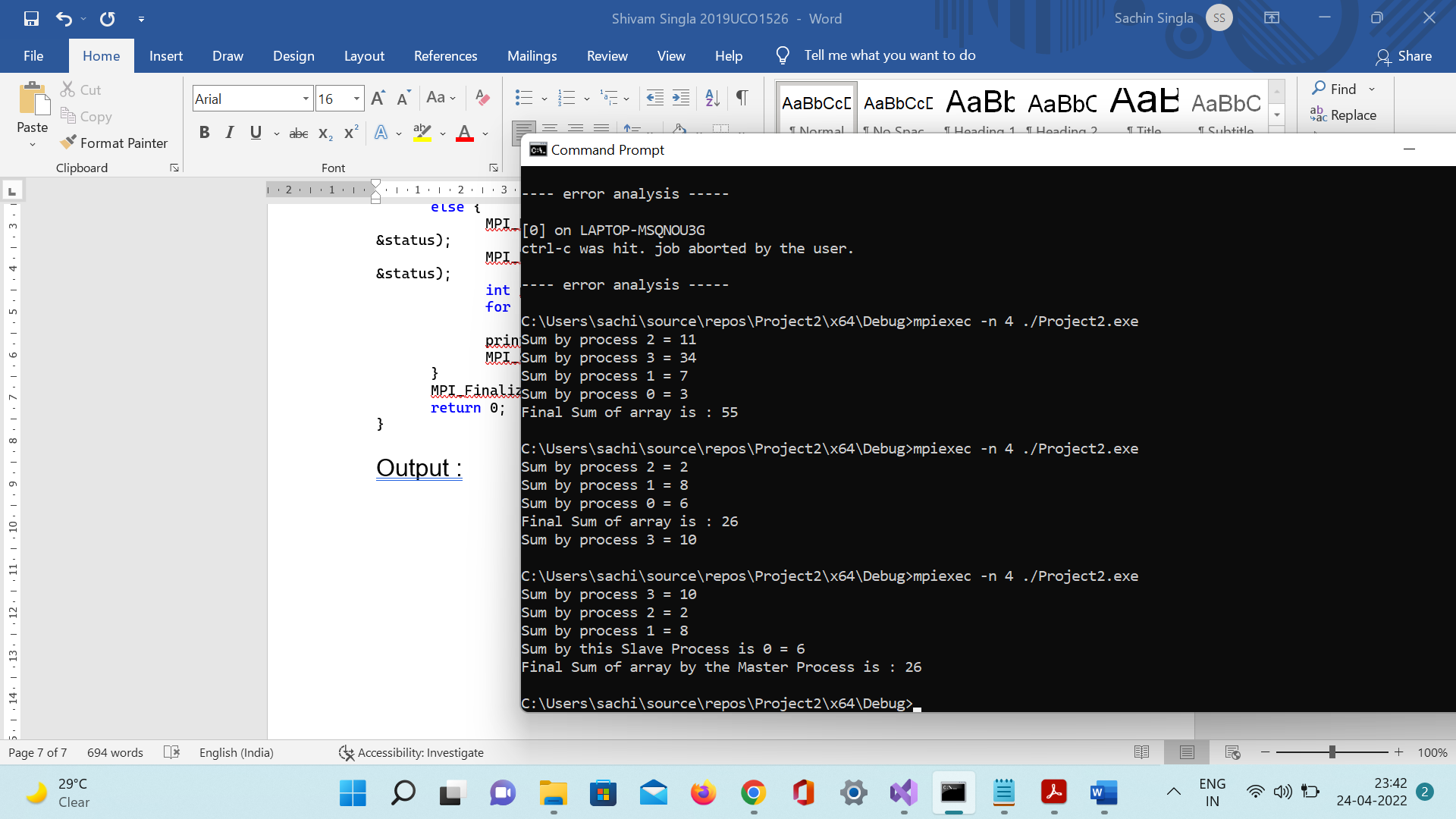
}

MPI\_Finalize();

return 0;

}

Output :



Experiment 4

Write a program to illustrate basic MPI communication routines

Code:

#include <mpi.h>

#include <stdio.h>

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

{

// Initialize the MPI environment

MPI\_Init(NULL, NULL);

// Get the number of processes

int world\_size;

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

// COMM\_WORLD is the communicator world

// Get the rank of the process

int world\_rank;

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

// Get the name of the processor

char processor\_name[MPI\_MAX\_PROCESSOR\_NAME];

int name\_len;

MPI\_Get\_processor\_name(processor\_name, &name\_len);

printf("Hello world from process %s, rank %d out of %d processes\n\n",

processor\_name, world\_rank, world\_size);

if (world\_rank == 0)

{

char message[] = "Shivam";

MPI\_Send(message, 6, MPI\_CHAR, 1, 0, MPI\_COMM\_WORLD);

}

else

{

char message[6];

MPI\_Recv(message, 6, MPI\_CHAR, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

printf("Message Successfully Received\n");

printf("Message Recieved : %s\n", message);

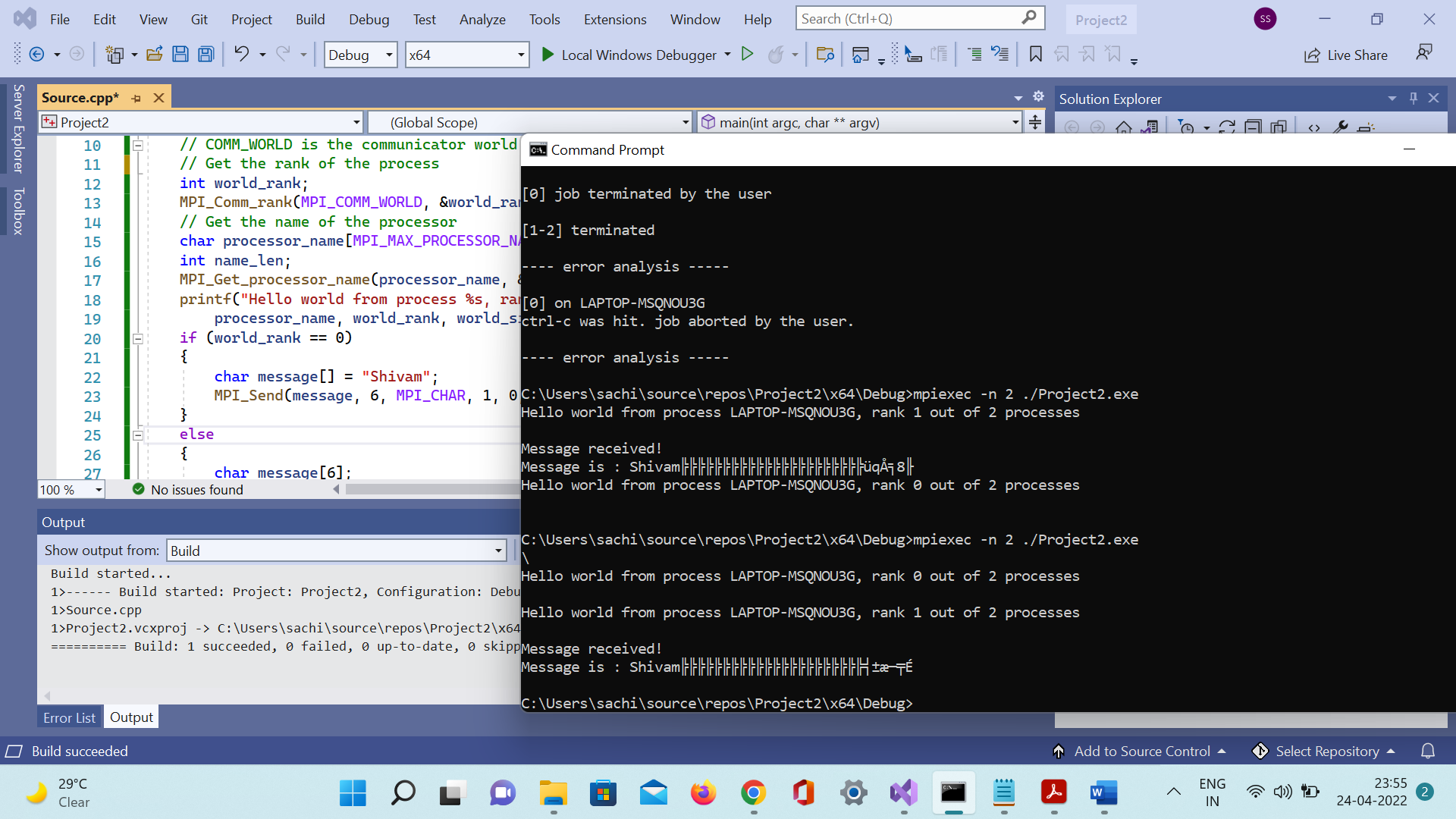
}

MPI\_Finalize();

return 0;

}

Output:



Experiment 5

Design a parallel program for summing up an array, matrix multiplication and show logging and tracing MPI activity

Code:

#include<stdio.h>

#include<iostream>

#include "mpi.h"

#define NUM\_ROWS\_A 8

#define NUM\_COLUMNS\_A 10

#define NUM\_ROWS\_B 10

#define NUM\_COLUMNS\_B 8

#define MASTER\_TO\_SLAVE\_TAG 1 //tag for messages sent from master to slaves

#define SLAVE\_TO\_MASTER\_TAG 4 //tag for messages sent from slaves to master

void create\_matrix();

void printArray();

int rank;

int size;

int i, j, k;

double A[NUM\_ROWS\_A][NUM\_COLUMNS\_A];

double B[NUM\_ROWS\_B][NUM\_COLUMNS\_B];

double result[NUM\_ROWS\_A][NUM\_COLUMNS\_B];

int low\_bound; //low bound of the number of rows of [A] allocated to a slave

int upper\_bound; //upper bound of the number of rows of [A] allocated to a slave

int portion; //portion of the number of rows of [A] allocated to a slave

MPI\_Status status; // store status of a MPI\_Recv

MPI\_Request request; //capture request of a MPI\_Send

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

{

MPI\_Init(&argc, &argv);

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

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

if (rank == 0)

{ // master process

create\_matrix();

for (i = 1; i < size; i++)

{

portion = (NUM\_ROWS\_A / (size - 1)); // portion without master

low\_bound = (i - 1) \* portion;

if (((i + 1) == size) && ((NUM\_ROWS\_A % (size - 1)) != 0))

{//if rows of [A] cannot be equally divided among slaves

upper\_bound = NUM\_ROWS\_A; //last slave gets all the remaining rows

}

else {

upper\_bound = low\_bound + portion; //rows of [A] are equally divisable among slaves

}

MPI\_Send(&low\_bound, 1, MPI\_INT, i, MASTER\_TO\_SLAVE\_TAG,

MPI\_COMM\_WORLD);

MPI\_Send(&upper\_bound, 1, MPI\_INT, i, MASTER\_TO\_SLAVE\_TAG + 1,

MPI\_COMM\_WORLD);

MPI\_Send(&A[low\_bound][0], (upper\_bound - low\_bound) \* NUM\_COLUMNS\_A,

MPI\_DOUBLE, i, MASTER\_TO\_SLAVE\_TAG + 2, MPI\_COMM\_WORLD);

}

}

//broadcast [B] to all the slaves

MPI\_Bcast(&B, NUM\_ROWS\_B \* NUM\_COLUMNS\_B, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

/\* Slave process\*/

if (rank > 0)

{

MPI\_Recv(&low\_bound, 1, MPI\_INT, 0, MASTER\_TO\_SLAVE\_TAG, MPI\_COMM\_WORLD,

&status);

MPI\_Recv(&upper\_bound, 1, MPI\_INT, 0, MASTER\_TO\_SLAVE\_TAG + 1,

MPI\_COMM\_WORLD, &status);

MPI\_Recv(&A[low\_bound][0], (upper\_bound - low\_bound) \* NUM\_COLUMNS\_A,

MPI\_DOUBLE, 0, MASTER\_TO\_SLAVE\_TAG + 2, MPI\_COMM\_WORLD, &status);

printf("Process %d calculating for rows %d to %d of Matrix A\n", rank,

low\_bound, upper\_bound);

for (i = low\_bound; i < upper\_bound; i++)

{

for (j = 0; j < NUM\_COLUMNS\_B; j++)

{

for (k = 0; k < NUM\_ROWS\_B; k++)

{

result[i][j] += (A[i][k] \* B[k][j]);

}

}

}

MPI\_Send(&low\_bound, 1, MPI\_INT, 0, SLAVE\_TO\_MASTER\_TAG, MPI\_COMM\_WORLD);

MPI\_Send(&upper\_bound, 1, MPI\_INT, 0, SLAVE\_TO\_MASTER\_TAG + 1,

MPI\_COMM\_WORLD);

MPI\_Send(&result[low\_bound][0], (upper\_bound - low\_bound) \* NUM\_COLUMNS\_B,

MPI\_DOUBLE, 0, SLAVE\_TO\_MASTER\_TAG + 2, MPI\_COMM\_WORLD);

}

/\* master gathers processed work\*/

if (rank == 0) {

for (i = 1; i < size; i++) {

MPI\_Recv(&low\_bound, 1, MPI\_INT, i, SLAVE\_TO\_MASTER\_TAG, MPI\_COMM\_WORLD,

&status);

MPI\_Recv(&upper\_bound, 1, MPI\_INT, i, SLAVE\_TO\_MASTER\_TAG + 1,

MPI\_COMM\_WORLD, &status);

MPI\_Recv(&result[low\_bound][0], (upper\_bound - low\_bound) \*

NUM\_COLUMNS\_B, MPI\_DOUBLE, i, SLAVE\_TO\_MASTER\_TAG + 2, MPI\_COMM\_WORLD, &status);

}

printArray();

}

MPI\_Finalize();

return 0;

}

void create\_matrix()

{

for (i = 0; i < NUM\_ROWS\_A; i++) {

for (j = 0; j < NUM\_COLUMNS\_A; j++) {

A[i][j] = i + j;

}

}

for (i = 0; i < NUM\_ROWS\_B; i++) {

for (j = 0; j < NUM\_COLUMNS\_B; j++) {

B[i][j] = i \* j;

}

}

}

void printArray()

{

printf("Given matrix A is: \n");

for (i = 0; i < NUM\_ROWS\_A; i++) {

printf("\n");

for (j = 0; j < NUM\_COLUMNS\_A; j++)

printf("%8.2f ", A[i][j]);

}

printf("\n\n\n");

printf("Given matrix B is: \n");

for (i = 0; i < NUM\_ROWS\_B; i++) {

printf("\n");

for (j = 0; j < NUM\_COLUMNS\_B; j++)

printf("%8.2f ", B[i][j]);

}

printf("\n\n\n");

printf("Final Multiplied Matrix is: \n");

for (i = 0; i < NUM\_ROWS\_A; i++) {

printf("\n");

for (j = 0; j < NUM\_COLUMNS\_B; j++)

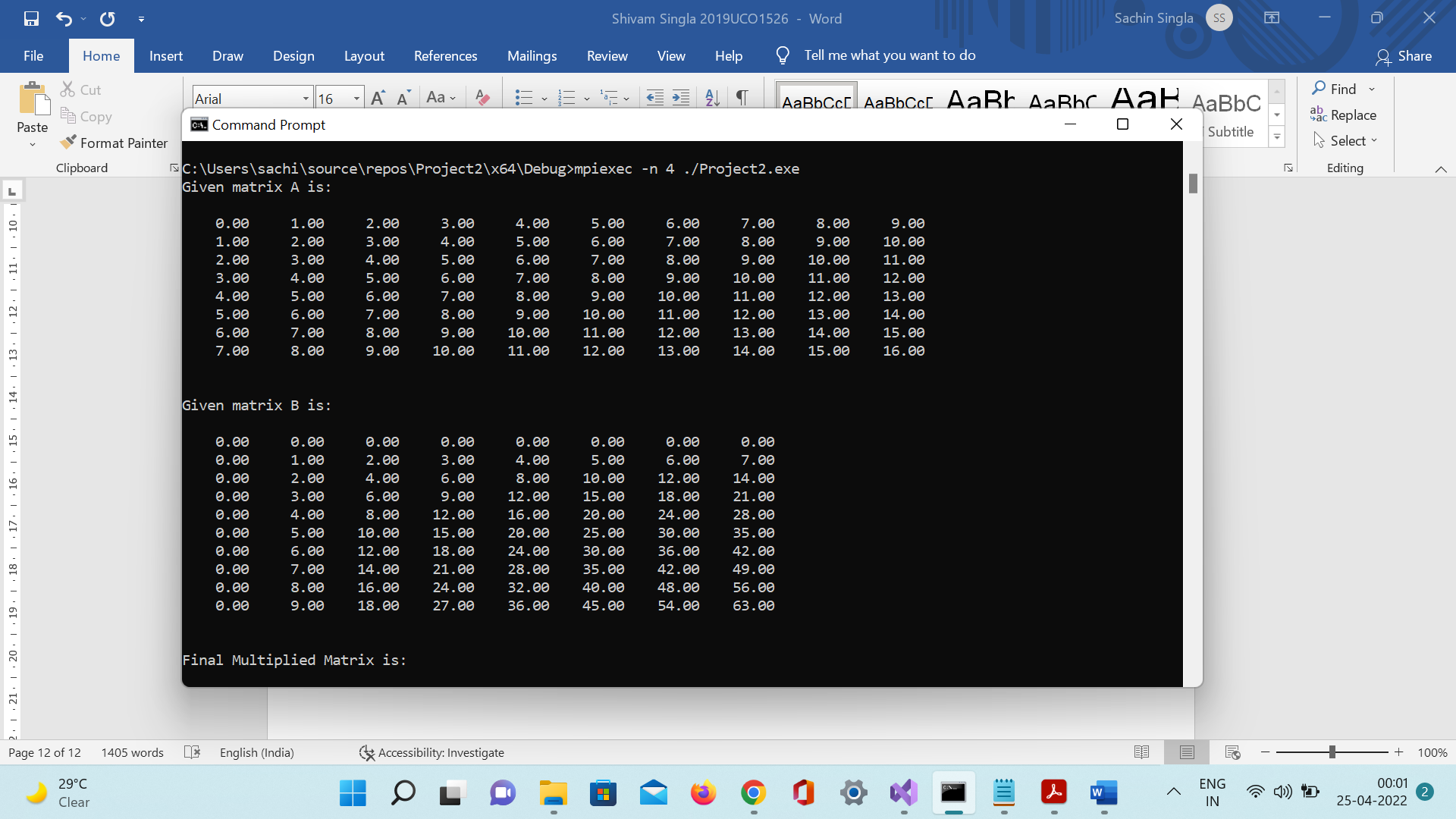
printf("%8.2f ", result[i][j]);

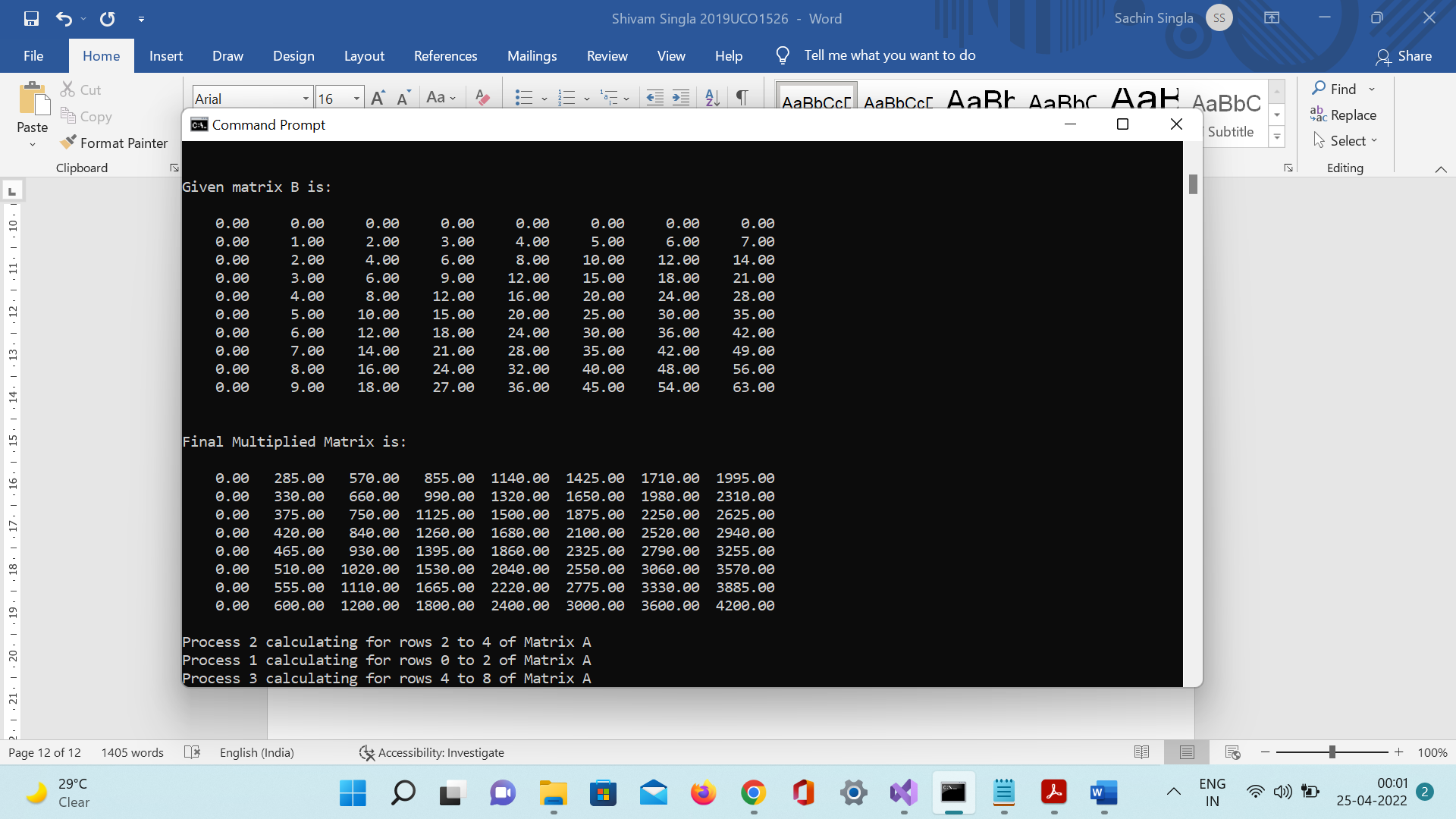
}

printf("\n\n");

}

Output:





Experiment 6

Write a C program with openMP to implement loop work sharing

Code:

#include <omp.h>

#include <stdio.h>

#include <iostream>

using namespace std;

void reset\_freq(int\* freq, int THREADS)

{

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

freq[i] = 0;

}

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

{

int n, THREADS, i;

printf("Enter the number of iterations :");

scanf\_s("%d", &n);

printf("Enter the number of threads (max 8): ");

scanf\_s("%d", &THREADS);

int freq[6];

reset\_freq(freq, THREADS);

// simple parallel for with unequal iterations

#pragma omp parallel for num\_threads(THREADS)

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

{

// printf("Thread num %d executing iter %d\n", omp\_get\_thread\_num(), i);

freq[omp\_get\_thread\_num()]++;

}

#pragma omp barrier

printf("\nIn default scheduling, we have the following thread distribution :- \n");

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

{

printf("Thread No. %d : %d iters\n", i, freq[i]);

}

// using static scheduling

int CHUNK;

printf("\nUsing static scheduling...\n");

printf("Enter the chunk size :");

scanf\_s("%d", &CHUNK);

// using a static, round robin schedule for the loop iterations

reset\_freq(freq, THREADS);

// useful when the workload is ~ same across each thread, not when otherwise

#pragma omp parallel for num\_threads(THREADS) schedule(static, CHUNK)

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

{

// printf("Thread num %d executing iter %d\n", omp\_get\_thread\_num(), i);

freq[omp\_get\_thread\_num()]++;

}

#pragma omp barrier

printf("\nIn static scheduling, we have the following thread distribution :- \n");

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

{

printf("Thread No. %d : %d iterations\n", i, freq[i]);

}

// auto scheduling depending on the compiler

printf("\nUsing automatic scheduling...\n");

reset\_freq(freq, THREADS);

#pragma omp parallel for num\_threads(THREADS) schedule(static)

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

{

// printf("Thread num %d executing iter %d\n", omp\_get\_thread\_num(), i);

freq[omp\_get\_thread\_num()]++;

}

#pragma omp barrier

printf("In auto scheduling, we have the following thread distribution :- \n");

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

{

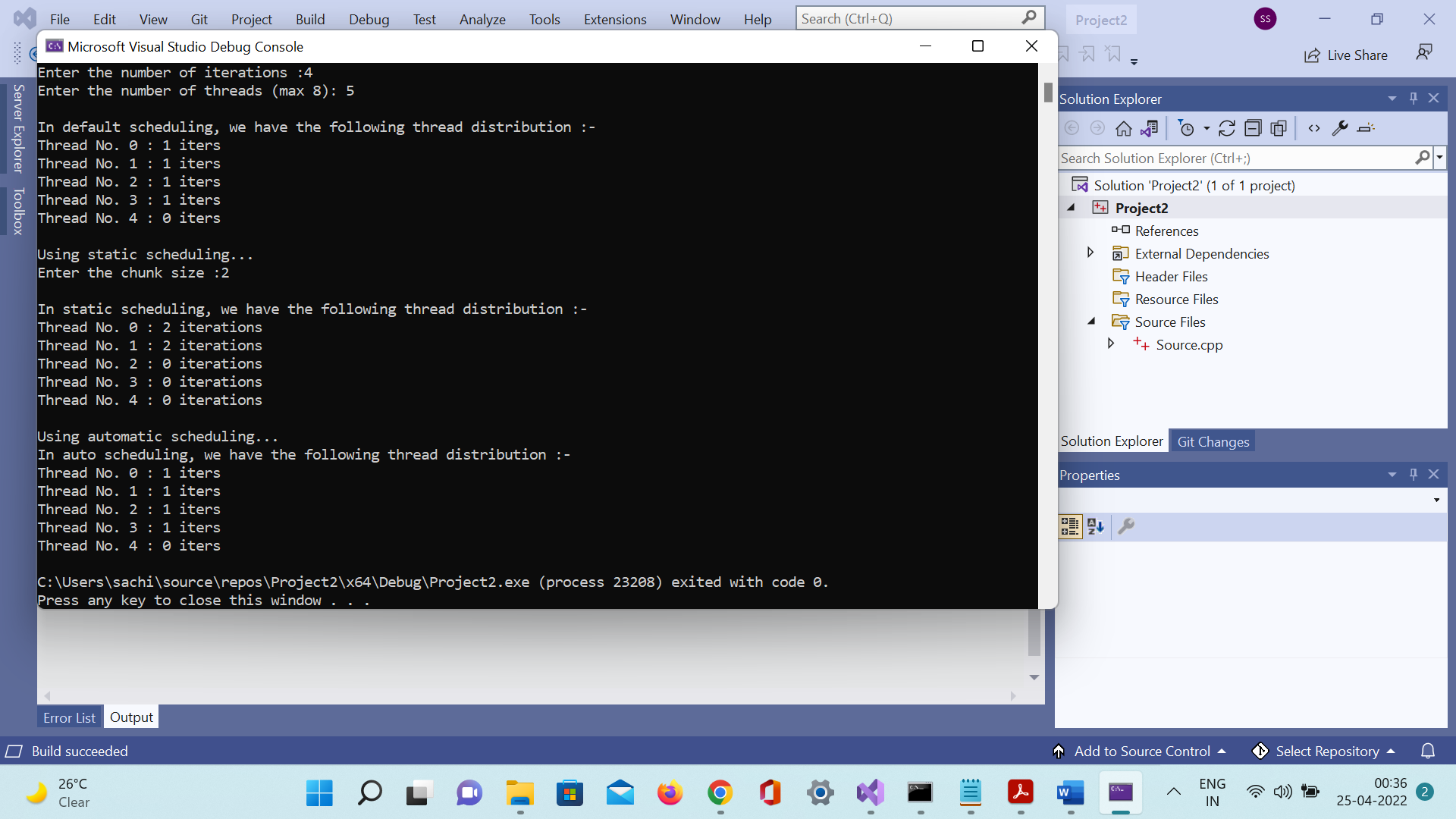
printf("Thread No. %d : %d iters\n", i, freq[i]);

}

return 0;

}

Output:



Experiment 7

Write a C program with openMP to implement loop work sharing

Code:

#include <omp.h>

#include <stdio.h>

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

{ // invocation of the main program

// use the fopenmp flag for compiling

int num\_threads, THREAD\_COUNT = 4;

int thread\_ID;

int section\_sizes[4] = {

0, 100, 200, 300 };

printf("Implementing Work load sharing of threads...\n");

#pragma omp parallel private(thread\_ID) num\_threads(THREAD\_COUNT)

{

// private means each thread will have a private variable

// thread\_ID

thread\_ID = omp\_get\_thread\_num();

printf("I am thread number %d!\n", thread\_ID);

int value\_count = 0;

if (thread\_ID > 0)

{

int work\_load = section\_sizes[thread\_ID];

// each thread has a different section size

for (int i = 0; i < work\_load; i++)

value\_count++;

printf("Total Number of values computed are : %d\n", value\_count);

}

#pragma omp barrier

if (thread\_ID == 0)

{

printf("The Total number of threads are : %d", omp\_get\_num\_threads());

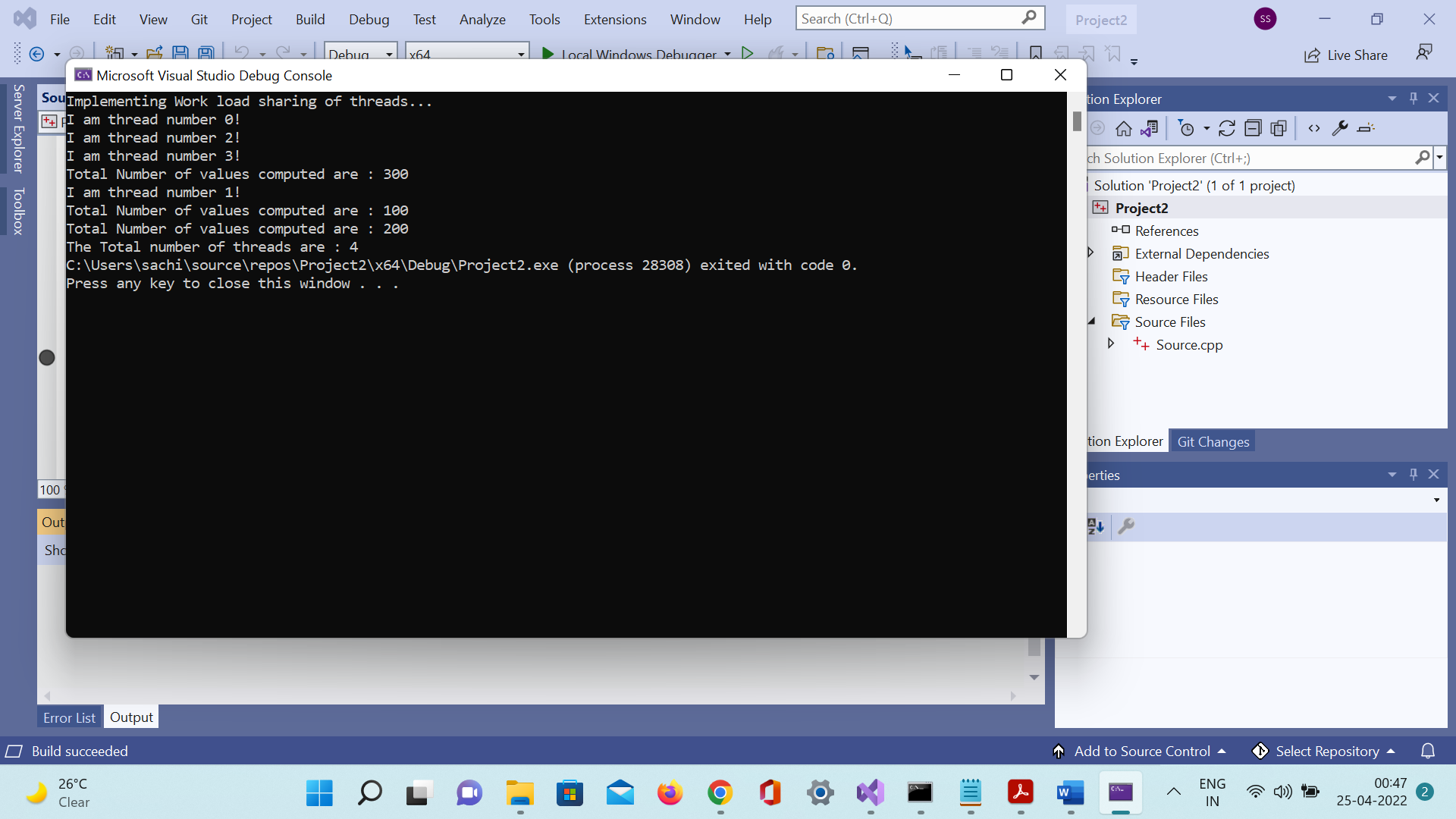
}

}

return 0;

}

Output:



Experiment 8

Write a program to illustrate process synchronization and collective data movements

Code:

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int thread\_count; // this global variable is shared by all threads

// compiling information -

// gcc name\_of\_file.c -o name\_of\_exe -lpthread (link p thread)

// necessary for referencing in the thread

struct arguments

{

int size;

int\* arr1;

int\* arr2;

int\* dot;

};

// function to parallelize`

void\* add\_into\_one(void\* arguments);

// util

void print\_vector(int n, int\* arr)

{

printf("[ ");

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

22

printf("%d ", arr[i]);

printf("] \n");

}

// main driver function of the program

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

{

long thread;

// /∗ Use long in case of a 64−bit system ∗/

pthread\_t\* thread\_handles;

thread\_count = 2; // using 2 threads only

// get the thread handles equal to total num

// of threads

thread\_handles = malloc(thread\_count \* sizeof(pthread\_t));

printf("Enter the size of the vectors : ");

int n;

scanf("%d", &n);

printf("Enter the max\_val of the vectors : ");

int max\_val;

scanf("%d", &max\_val);

struct arguments\* args[2]; // array of pointer to structure

// each element is a pointer

for (int i = 0; i < 2; i++)

{

// allocate for the struct

args[i] = malloc(sizeof(struct arguments) \* 1);

// allocate for the arrays

args[i]->size = n;

args[i]->arr1 = malloc(sizeof(int) \* n);

args[i]->arr2 = malloc(sizeof(int) \* n);

args[i]->dot = malloc(sizeof(int) \* n);

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

23

{

args[i]->arr1[j] = rand() % max\_val;

args[i]->arr2[j] = rand() % max\_val;

}

}

printf("Vectors are : \n");

print\_vector(n, args[0]->arr1);

print\_vector(n, args[0]->arr2);

print\_vector(n, args[1]->arr1);

print\_vector(n, args[1]->arr2);

int result[n];

memset(result, 0, n \* sizeof(int));

// note : we need to manually startup our threads

// for a particular function which we want to execute in

// the thread

for (thread = 0; thread < thread\_count; thread++)

{

printf("Multiplying %ld and %ld with thread %ld...\n", thread + 1, thread + 2,

thread);

pthread\_create(&thread\_handles[thread], NULL, add\_into\_one, (void\*)args[thread]);

}

printf("Hello from the main thread\n");

// wait for completion

for (thread = 0; thread < thread\_count; thread++)

pthread\_join(thread\_handles[thread], NULL);

for (int i = 0; i < 2; i++)

{

printf("Multiplication for vector %d and %d \n", i + 1, i + 2);

print\_vector(n, args[i]->dot);

printf("\n");

}

free(thread\_handles);

// now compute the summation of results

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

24

result[i] = args[0]->dot[i] + args[1]->dot[i];

printf("Result is : \n");

print\_vector(n, result);

return 0;

}

void\* add\_into\_one(void\* argument)

{

// de reference the argument

struct arguments\* args = argument;

// compute the dot product into the

// array dot

int n = args->size;

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

args->dot[i] = args->arr1[i] \* args->arr2[i];

return NULL;

}

Output:

