Parallel Computing Lab
Nilay Ganvit - 200001053
30th November 2022

Assignment

Q1.

```
//including headers
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <mpi.h>
#include <time.h>
//Driver Code
int main(int argc, char **argv){

   int rank_usr, proc_num;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &proc_num);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank_usr);
   MPI_Request req[proc_num + 1];
   MPI_Status status;
   int num;
   int Rec_Dat[3];
   int Send_Dat[proc_num + 1][3];
   int filled[1000];
   int weights[1000];
   int mat_graph[1000][1000];
   int num_max_colour = 0;
   int vertices, edges;
   //Initiallising filled array
   int l = 0;
   for (l = 0; l < 1000; l++)
       filled[l] = -1;
   //Master Node
   if (rank_usr == 0)
   {   //taking input
       scanf("%d", &vertices);
       scanf("%d", &edges);
       int i, j;
   //generating random weights
```

```
        for (i = 0; i < vertices; i++)
        {
            weights[i] = rand() % 42 + rand() % 42 + rand() % 42 + rand() %
42;
            filled[i] = -1;
        }
        //taking parameters input
        int start, end;
        for (i = 0; i < edges; i++)
        {
            scanf("%d %d", &start, &end);
            mat_graph[start][end] = 1;
            mat_graph[end][start] = 1;
        }
        //Recording time
        clock_t begin = clock();
        int max = -1;
        //initial iterrations by the master node
        for (i = 0; i < vertices; i++)
        {
            max = 0;
            for (j = 0; j < vertices; j++)
            {
                if (mat_graph[i][j] != 0)
                    max++;
            }
            if (max > num_max_colour)
                num_max_colour = max;
        }

        num_max_colour += 1;
        //diving and allocating node among worksers
        int num_vetices_per_proc = vertices / (proc_num - 1);
        int num_vetices_remaining = vertices / (proc_num - 1) + (vertices %
(proc_num - 1));
        int startVertex = 0, endVertex;
        //Sending data to the worker nodes
        for (i = 1; i < proc_num - 1; i++)
        {
            MPI_Send(&vertices, 1, MPI_INT, i, i, MPI_COMM_WORLD);
```

```
            int k = 0;
            for (k = 0; k < vertices; k++)
            {
                MPI_Send(&mat_graph[k], vertices, MPI_INT, i, i,
MPI_COMM_WORLD);
            }

            endVertex = startVertex + num_vetices_per_proc - 1;
            Send_Dat[i][0] = startVertex;
            Send_Dat[i][1] = endVertex;
            Send_Dat[i][2] = num_max_colour;
            MPI_Send(Send_Dat[i], 3, MPI_INT, i, i, MPI_COMM_WORLD);
            MPI_Send(filled, vertices, MPI_INT, i, i, MPI_COMM_WORLD);
            MPI_Send(weights, vertices, MPI_INT, i, i, MPI_COMM_WORLD);
            startVertex = endVertex + 1;
        }
        MPI_Send(&vertices, 1, MPI_INT, i, i, MPI_COMM_WORLD);

        int k = 0;
        for (k = 0; k < vertices; k++)
            MPI_Send(&mat_graph[k], vertices, MPI_INT, i, i,
MPI_COMM_WORLD);

        endVertex = startVertex + num_vetices_remaining - 1;
        Send_Dat[i][0] = startVertex;
        Send_Dat[i][1] = endVertex;
        Send_Dat[i][2] = num_max_colour;
        MPI_Send(Send_Dat[i], 3, MPI_INT, i, i, MPI_COMM_WORLD);
        MPI_Send(filled, vertices, MPI_INT, i, i, MPI_COMM_WORLD);
        MPI_Send(weights, vertices, MPI_INT, i, i, MPI_COMM_WORLD);
        //recieving final calculations from the worker nodes
        for (i = 1; i <= proc_num - 1; i++)
            MPI_Recv(filled, vertices, MPI_INT, i, i, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

        max = -1;
        for (i = 0; i < vertices; i++)
        {
            if (filled[i] > max)
                max = filled[i];
```

```
        }
        printf("%d\n", max + 1);
        for (i = 0; i < vertices; i++)
            printf("%d\n", filled[i]);
        //printing runtime
        clock_t final = clock();
        printf("Time taken in seconds is: %lf\n", ((double)(final/8 -
begin/8) /CLOCKS_PER_SEC));
    }
    else //worker nodes
    {   //recieving data from master node
        int vertices;
        MPI_Recv(&vertices, 3, MPI_INT, 0, rank_usr, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

        int mat_graph[vertices][vertices];

        int i, j;
        for (i = 0; i < vertices; i++)
            MPI_Recv(&mat_graph[i], vertices, MPI_INT, 0, rank_usr,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        MPI_Request receiver_req;
        MPI_Status receiver_status;

        MPI_Recv(Rec_Dat, 3, MPI_INT, 0, rank_usr, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        int filled[vertices], weights[vertices];
        MPI_Recv(filled, vertices, MPI_INT, 0, rank_usr, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        MPI_Recv(weights, vertices, MPI_INT, 0, rank_usr, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        //executing the greedy bfs algorithm
        int vertex_curr;
        int round = 0;
        while (round < vertices) //outer loop
        {
            int all_filled = 1;
            for (vertex_curr = Rec_Dat[0]; vertex_curr <= Rec_Dat[1];
vertex_curr++) //inner loop
```

```
            {
                if (filled[vertex_curr] == -1)
                {
                    all_filled = 0;

                    int weight_usr = weights[vertex_curr];

                    int max_wt_usr = 1;

                    int adjacent;

                    for (adjacent = 0; adjacent < vertices; adjacent++)
//caculating if adj isn't current vertex
                    {
                        if (adjacent != vertex_curr)
                        {
                            if (mat_graph[vertex_curr][adjacent] == 1 &&
filled[adjacent] == -1 && weights[adjacent] > weights[vertex_curr])
                            {
                                max_wt_usr = 0;
                                break;
                            }
                        }
                    }
                    if (max_wt_usr)
                    {
                        int colour_filled = -1;

                        for (colour_filled = 0; colour_filled < Rec_Dat[2];
colour_filled++)
                        {
                            int avail_col = 1;
                            for (adjacent = 0; adjacent < vertices;
adjacent++)
                            {
                                if (mat_graph[vertex_curr][adjacent] == 1
&& filled[adjacent] != -1 && colour_filled == filled[adjacent])
                                {
                                    avail_col = 0;
                                    break;
```

```
                                }
                            }
                            if (avail_col == 1)
                            {
                                filled[vertex_curr] = colour_filled;
                                break;
                            }
                        }
                    }
                }
            }
            //Sending calculated parameters
            int m;
            for (m = 1; m <= proc_num - 1; m++)
            {
                if (m != rank_usr)
                    MPI_Send(filled, vertices, MPI_INT, m, m,
MPI_COMM_WORLD);
            }
            //recieving calculated parameters for next iteration
            for (m = 1; m <= proc_num - 1; m++)
            {
                int col_usr[vertices], weight_curr_usr[vertices];
                if (m != rank_usr)
                    MPI_Recv(col_usr, vertices, MPI_INT, m, rank_usr,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

                int count = 0;
                for (count = 0; count < vertices; count++)
                {
                    if (filled[count] == -1 && col_usr[count] != -1 &&
col_usr[count] < Rec_Dat[2] && col_usr[count] >= 0)
                        filled[count] = col_usr[count];
                }
            }
            int count;
            round++;
            sleep(3);
        }
```

```
        MPI_Send(filled, vertices, MPI_INT, 0, rank_usr, MPI_COMM_WORLD);
//sneding data to master node
    }
   MPI_Finalize();
   return 0;
}
```

Input/Output:

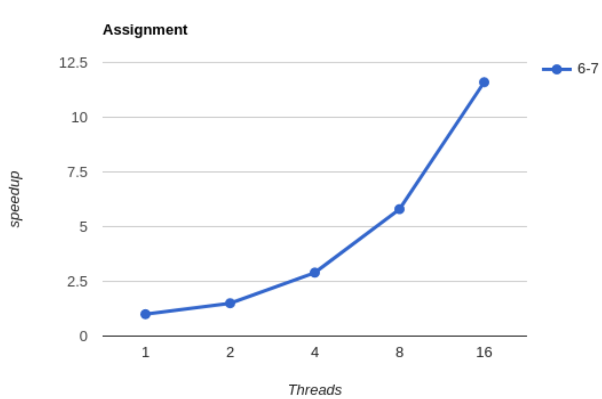
```
nilay@Nilay-PC:~/Documents/cs359/Assignment$ mpiexec -n 2 ./mpi
6 7
0 1
0 2
1 3
2 3
3 4
3 5
4 5
3
0
0
1
0
1
2
Time taken in seconds is: 17.995198
nilay@Nilay-PC:~/Documents/cs359/Assignment$
```

```
nilay@Nilay-PC:~/Documents/cs359/Assignment$ mpiexec -n 4 ./mpi
6 7
0 1
0 2
1 3
2 3
3 4
3 5
4 5
3
0
1
1
0
1
2
Time taken in seconds is: 8.992096
nilay@Nilay-PC:~/Documents/cs359/Assignment$
```

```
nilay@Nilay-PC:~/Documents/cs359/Assignment$ mpiexec -n 8 ./mpi
6 7
0 1
0 2
1 3
2 3
3 4
3 5
4 5
3
0
1
1
0
1
2
Time taken in seconds is: 4.519935
nilay@Nilay-PC:~/Documents/cs359/Assignment$
```

```
nilay@Nilay-PC:~/Documents/cs359/Assignment$ mpiexec -n 16 ./mpi
6 7
0 1
0 2
1 3
2 3
3 4
3 5
4 5
3
0
1
1
0
1
2
Time taken in seconds is: 2.280828
nilay@Nilay-PC:~/Documents/cs359/Assignment$
```

Graph for Speedup: 1 core-1, 2 core~1.51, 4 core~2.92 8 core~5.83, 16 core~11.64



Description:
The algorithm used here is basically a greedy approach to the problem.
Create every conceivable combination of colors. The total number of possible color combinations is mV since each node can be coloured using any of the m-accessible colors. Verify whether or not the adjacent vertices share the same color after generating a color configuration. Print the combination and end the loop if the conditions are satisfied.

To solve the issue, adhere to the suggested instructions.

- Make a loop that accepts the output color array, the number of vertices, and the current index.
- Whether the number of vertices and the current index match. Verify that the output color configuration is secure. That is, ensure that no two neighboring vertices have the same color. Print the settings and break if the conditions are met.
- Give a vertex a color (1 to m).
- Loop for each color assigned, passing it to the following index and the number of vertices.
- Break the loop and return true if any loop returns true.

Inference & Parallel Exploitation:
While the outermost loops can't parallelise because of communication overheads the inner loops can be run in parallel as the overheads then will weigh in as much as the other calculations. And the Program seems to give diminishing speedup which is common.

Q2.

```
//Including Header files
#include <stdio.h>
#include <mpi.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#define MAX 50

//DFS
int depth_search(int begin, int n, int mat_graph[MAX][MAX], int k)
{
   if (k == 0)
       return 1;

   int cnt_tot = 0, i = 1;
```

```
    for (; i <= n; ++i)
    {
        if (mat_graph[begin][i] == 1)
            cnt_tot += depth_search(i, n, mat_graph, k - 1);
    }

    return cnt_tot;
}
//Main function
int main(int argc, char **argv)
{

    int proc_num, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &proc_num);
    MPI_Status status;

    int flag = 5, flag1 = 10;
    int walk_tot = 0;
    int mat_graph[MAX][MAX] = {0};
    // Master Node
    if (rank == 0)
    {
        int n, e, k;
        scanf("%d %d %d", &n, &e, &k);

        // Randomly generating graph
        int s, d;
        for (int i = 0; i < e; i++)
        {
            s=rand()%n;
            d=rand()%n;
            mat_graph[s][d] = 1;
        }
        // Allocating nodes for processes
        int num_node_on_proc = n / (proc_num - 1);
        int num_node_remain = n % (proc_num - 1);

        // setting bounds
```

```
        int boundl = 1, boundu = num_node_on_proc;

        if (num_node_remain != 0)
        {
            boundu = num_node_remain;

            // Master calculates offsets
            for (int node = boundl; node <= boundu; ++node)
                walk_tot += depth_search(node, n, mat_graph, k);

            boundl = boundu + 1;
            boundu = boundl + num_node_on_proc - 1;
        }

        // Communicating: Sending data to workers
        for (int dest = 1; dest < proc_num; dest++)
        {
            MPI_Send(&boundl, 1, MPI_INT, dest, flag, MPI_COMM_WORLD);
            MPI_Send(&boundu, 1, MPI_INT, dest, flag + 1, MPI_COMM_WORLD);
            MPI_Send(&k, 1, MPI_INT, dest, flag + 2, MPI_COMM_WORLD);
            MPI_Send(&n, 1, MPI_INT, dest, flag + 3, MPI_COMM_WORLD);
            boundl = boundu + 1;
            boundu += num_node_on_proc;
        }
    }

    clock_t start = clock();
    // Sending(Broadcast) graph to all nodes
    MPI_Bcast(&mat_graph, MAX * MAX, MPI_INT, 0, MPI_COMM_WORLD);

    // Worker process
    if (rank > 0)
    {

        // Receiving Data from Master Node
        int boundl, boundu, k, n;
        MPI_Recv(&boundl, 1, MPI_INT, 0, flag, MPI_COMM_WORLD, &status);
        MPI_Recv(&boundu, 1, MPI_INT, 0, flag + 1, MPI_COMM_WORLD,
&status);
        MPI_Recv(&k, 1, MPI_INT, 0, flag + 2, MPI_COMM_WORLD, &status);
```

```
        MPI_Recv(&n, 1, MPI_INT, 0, flag + 3, MPI_COMM_WORLD, &status);
        int walk_curr = 0;

        // Calculating walks
        for (int node = boundl; node <= boundu; node++)
        {
            int x = depth_search(node, n, mat_graph, k);
            walk_curr += x;
        }

        // Sending result data to master
        MPI_Send(&walk_curr, 1, MPI_INT, 0, flag1, MPI_COMM_WORLD);
    }
    // Master
    if (rank == 0)
    {

        // Receive walks from respective worker nodes
        int x;
        for (int src = 1; src < proc_num; ++src)
        {
            MPI_Recv(&x, 1, MPI_INT, src, flag1, MPI_COMM_WORLD, &status);
            walk_tot += x;
        }
        printf("%d\n", walk_tot);
        clock_t end = clock();
        printf("Time taken in seconds is: %lf\n", ((double)(end - start)
/CLOCKS_PER_SEC));

    }
    MPI_Finalize();
    return 0;
}
```

Input/Output:

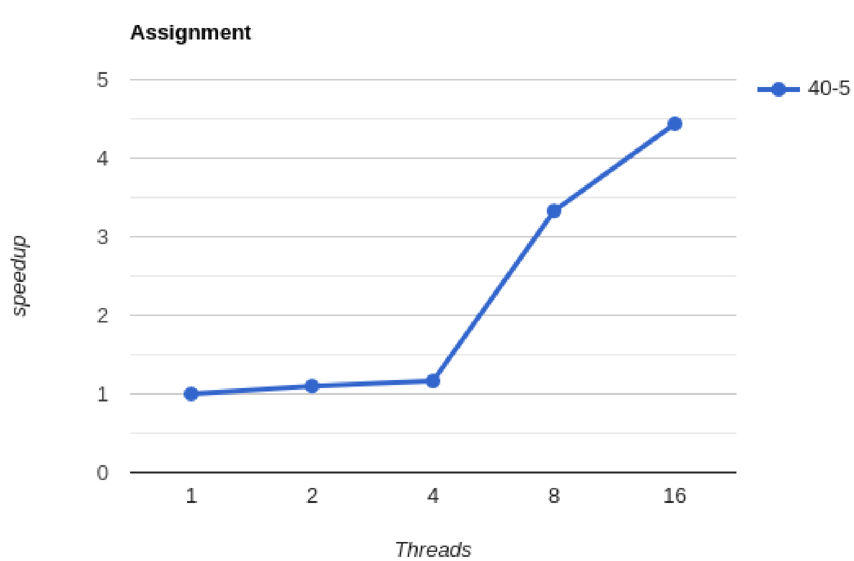
```
nilay@Nilay-PC:~/Documents/cs359/Assignment$ mpiexec -n 2 ./mpi
40 40 5
27
Time taken in seconds is: 0.000460
nilay@Nilay-PC:~/Documents/cs359/Assignment$
```

```
nilay@Nilay-PC:~/Documents/cs359/Assignment$ mpiexec -n 4 ./mpi
40 40 5
27
Time taken in seconds is: 0.000434
nilay@Nilay-PC:~/Documents/cs359/Assignment$
```

```
nilay@Nilay-PC:~/Documents/cs359/Assignment$ mpiexec -n 8 ./mpi
40 40 5
27
Time taken in seconds is: 0.000152
nilay@Nilay-PC:~/Documents/cs359/Assignment$
```

```
nilay@Nilay-PC:~/Documents/cs359/Assignment$ mpiexec -n 16 ./mpi
40 40 5
27
Time taken in seconds is: 0.000114
nilay@Nilay-PC:~/Documents/cs359/Assignment$
```

Graph:1 core-1, 2 core~1.1, 4 core~1.16, 8 core~3.328, 16 core~4.43



Description:
The algorithm is basically Depth First Search or DFS

- From the source vertex, perform a depth-first search, counting the edges along the way.

- Check the node to see if you have reached the 'v'th node after traversing the K-edge; otherwise, leave that path and look for alternative options.
- Similarly, print the queue and look for any further pathways that might exist if we observe the 'v th node after the 'k' edges.

Inference & Parallel Exploitation:
The Master Node first takes in input to randomly generate a graph. Then it allocates nodes to each process equally and remaining to the last one. After that it sets bounds and calculates the offsets. Lastly it sends all the data to the workers.
The Graph is Broadcasted to all the nodes.
The Worker Nodes receive data from the master node and the calculate walks for the given data and send the calculated result to the master
Lastly the master receives walks from the worker nodes and Calculates the final walks.
It can be seen that Better speedups aren’t received till between 4 & 8 processors. This might be due to not enough parallelisation in 2 or 4 cores.

Q3.

```
//Importing headers
#include <bits/stdc++.h>
#include <time.h>
#include <pthread.h>
using namespace std;

//Size of matrix
#define max 512
//mutex lock
pthread_mutex_t mutex_g;
//global variables
int ans_final = 0, n, thread_num, n_on_thread;
//array
int array_g[10000][2] = {0};
//calculating function
void *thread_func(void *s)
{

   int *index = (int *)s;
   int index_usr = *(index);
   int ans_curr = 0;
   //calculating and adding all to get final answer
   for (int i = 0; i < n_on_thread; ++i)
   {
       if (index_usr + i < n)
```

```
        {

            pthread_mutex_lock(&mutex_g); //lock

            ans_final += ((array_g[index_usr + i][0] * array_g[index_usr +
i + 1][1]) - (array_g[index_usr + i][1] * array_g[index_usr + i + 1][0]));
            pthread_mutex_unlock(&mutex_g);
        }
    }

    pthread_exit(0);
}
//main function
int main(int argc, char **argv)
{

    pthread_t p_threads[max];
    pthread_attr_t attr;
    clock_t begin, end;

    pthread_attr_init(&attr);
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    pthread_mutex_init(&mutex_g, NULL);
    //size
    n = 10000;
    //initialing random
    for (int i = 0; i < n; ++i)
    {

        array_g[i][0] = rand() % 42;
        array_g[i][1] = rand() % 42;
    }
    array_g[n][0] = array_g[0][0];
    array_g[n][1] = array_g[0][1];
    //Start recording time
    begin = clock();
    double area_sequential = 0;
    //Calculating area
    for (int i = 0; i < n; ++i)
    {
```

```
        area_sequential += (array_g[i][0] * array_g[i + 1][1] -
array_g[i][1] * array_g[i + 1][0]);
    }
    end = clock();
    double time_sequential = (double(end - begin)) / CLOCKS_PER_SEC;
    cout << "Sequential Area = " << area_sequential / 2 << endl;
    cout << "Time taken for Serial = " << time_sequential << endl;

    cout << "Threads = ";
    //user input
    cin >> thread_num;
    n_on_thread = n / thread_num;

    int *index_usr = new int(0);
    int num_index[thread_num] = {0};
    //Start recording time
    begin = clock();
    //Allocation of nodes
    for (int i = 0; i < thread_num; ++i)
    {
        num_index[i] = *(index_usr);
        *(index_usr) += n_on_thread;
    }
    //parallel execution
    for (int i = 0; i < thread_num; ++i)
    {
        pthread_create(&p_threads[i], &attr, thread_func, (void
*)(&num_index[i]));
    }
    for (int i = 0; i < thread_num; ++i)
    {
        pthread_join(p_threads[i], NULL);
    }
    end = clock();
    double parallel_time = (double(end - begin)) / CLOCKS_PER_SEC;
    cout << "Area Parallel = " << double(ans_final / 2) << endl;
    cout << "Time taken by parallel = " << parallel_time << endl;
    cout << "Speedup = " << double(time_sequential / parallel_time) <<
endl;
}
```

Input/Output

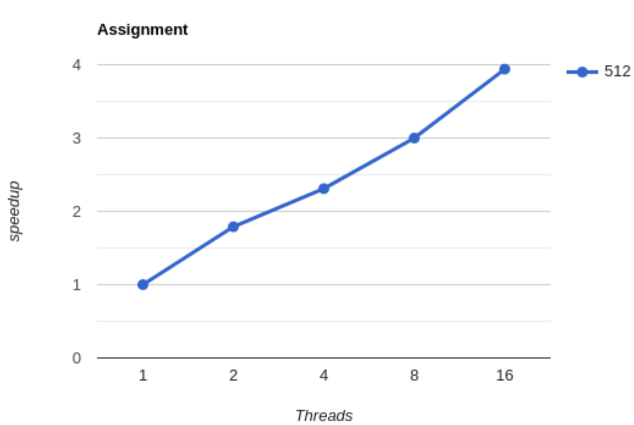
```
● nilay@Nilay-PC:~/Documents/cs359/Assignment$ ./a.out
  Sequential Area = -1328.5
  Time taken for Serial = 0.0089
  Threads = 2
  Area Parallel = -1328
  Time taken by parallel = 0.004956
  Speedup = 1.7958
```

```
● nilay@Nilay-PC:~/Documents/cs359/Assignment$ ./a.out
  Sequential Area = -1328.5
  Time taken for Serial = 0.0088
  Threads = 4
  Area Parallel = -1328
  Time taken by parallel = 0.003801
  Speedup = 2.31518
○ nilay@Nilay-PC:~/Documents/cs359/Assignment$
```

```
● nilay@Nilay-PC:~/Documents/cs359/Assignment$ ./a.out
  Sequential Area = -1328.5
  Time taken for Serial = 0.0088
  Threads = 8
  Area Parallel = -1328
  Time taken by parallel = 0.002925
  Speedup = 3.00855
○ nilay@Nilay-PC:~/Documents/cs359/Assignment$
```

```
● nilay@Nilay-PC:~/Documents/cs359/Assignment$ ./a.out
  Sequential Area = -1328.5
  Time taken for Serial = 0.0103
  Threads = 16
  Area Parallel = -1328
  Time taken by parallel = 0.002613
  Speedup = 3.94183
○ nilay@Nilay-PC:~/Documents/cs359/Assignment$
```

Speedup Graph: 1 core-1, 2 core~1.79, 4 core~2.31, 8 core~3.01, 16 core~3.94



Description:
A polygon can always be divided into triangles. The cross-product, which yields the area of a parallelogram, is taken, multiplied by 2, and used to calculate the (signed) area of the triangle with a vertex at the origin. This creates the area formula. These triangles with +ve and -ve areas will overlap as one loop around the polygon, canceling out and adding to zero the areas between the origin and the polygon, leaving only the area inside the reference triangle.

Area = $| 1/2 [(x_1y_2 + x_2y_3 + \ldots + x_{n-1}y_n + x_ny_1) – (x_2y_1 + x_3y_2 + \ldots + x_ny_{n-1} + x_1y_n)] |$

Inference & Parallel Exploitation:
Polygon Clipping is used as parallel exploitation includes division of the whole polygon in parts and calculating area of the parts and then finally adding them up.
An Almost Linear Graph is obtained for Speedup This shows the nature of the algorithm in mathematical form and how the ratio of work increment and core increment (in this case exponentially power of 2) from 2 to 4 to 8 to 16 results in a linear fashion of datapoints.

Q4.
```
//Header files needed for the program
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>

// Defining number of nodes
#define Num 1000

// Initializing matrix to 0
int mat_init[Num][Num] = {0};

int main(int argc, char *argv[])
{
   int thread_num;
   int iter, dist, pivot;

   // Taking random distances
   for (iter = 0; iter < Num; iter++)
   {
       for (dist = 0; dist < Num; dist++)
       {
```

```
            if (iter != dist)
            {
                // Generating random numbers
                mat_init[iter][dist] = rand() % 42;
            }
        }
    }

    // Start measuring time
    double start_time = omp_get_wtime();

    for (pivot = 0; pivot < Num; pivot++)
    {
        int *mat_ptr = mat_init[pivot];
        for (iter = 0; iter < Num; iter++)
        {
            int *dp = mat_init[iter];
            for (dist = 0; dist < Num; dist++)
            {
                dp[dist] = min(dp[dist], dp[pivot] + mat_ptr[dist]);
            }
        }
    }

    double time = omp_get_wtime() - start_time;
    printf("Sequential time = %.2f sec\n", time);
    double seqt=time;

    for (thread_num = 2; thread_num <= 16; thread_num=thread_num*2)
    {
        // Setting number of threads
        omp_set_num_threads(thread_num);

        // Start time for parallel execution
        double start_time = omp_get_wtime();

//Checking if smaller dist already present
#pragma omp parallel shared(mat_init)
        for (pivot = 0; pivot < Num; pivot++)
        {
```

```
            int *mat_ptr = mat_init[pivot];
#pragma omp parallel for private(iter, dist) schedule(dynamic)
            for (iter = 0; iter < Num; iter++)
            {
                int *dp = mat_init[iter];
                for (dist = 0; dist < Num; dist++)
                {
                    dp[dist] = min(dp[dist], dp[pivot] + mat_ptr[dist]);
                }
            }
        }

        double time = omp_get_wtime() - start_time;
        printf("parallel time for  %d threads = %.2f sec and speedup =
%2f\n", thread_num, time,seqt/time);
    }
    return 0;
}
```

Input/Output:

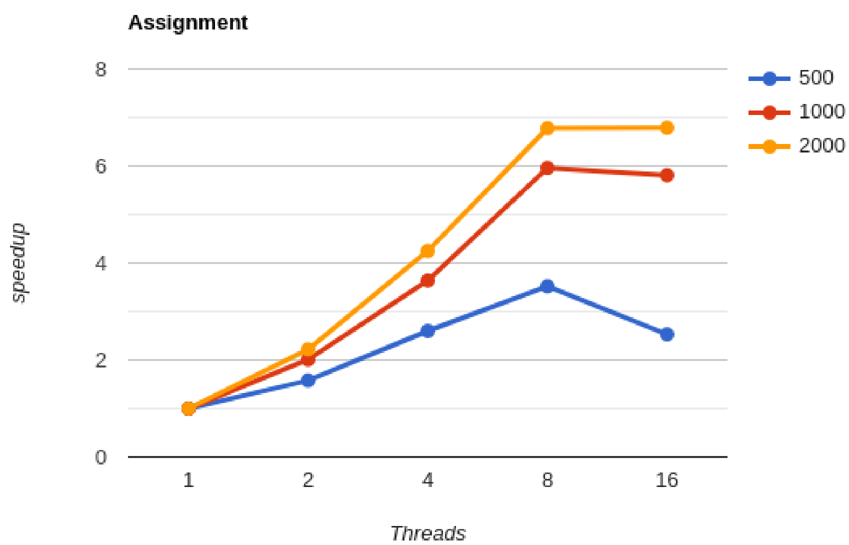
```
nilay@Nilay-PC:~/Documents/cs359/Assignment$ gcc -fopenmp Q4.c
nilay@Nilay-PC:~/Documents/cs359/Assignment$ ./a.out
Sequential time = 0.39 sec
parallel time for  2 threads = 0.25 sec and speedup = 1.583333
parallel time for  4 threads = 0.15 sec and speedup = 2.606855
parallel time for  8 threads = 0.11 sec and speedup = 3.522010
parallel time for  16 threads = 0.16 sec and speedup = 2.533321
nilay@Nilay-PC:~/Documents/cs359/Assignment$
```

```
nilay@Nilay-PC:~/Documents/cs359/Assignment$ gcc -fopenmp Q4.c
nilay@Nilay-PC:~/Documents/cs359/Assignment$ ./a.out
Sequential time = 5.58 sec
parallel time for  2 threads = 2.78 sec and speedup = 2.003086
parallel time for  4 threads = 1.53 sec and speedup = 3.643291
parallel time for  8 threads = 0.93 sec and speedup = 5.967423
parallel time for  16 threads = 0.96 sec and speedup = 5.806644
```

```
nilay@Nilay-PC:~/Documents/cs359/Assignment$ gcc -fopenmp Q4.c
nilay@Nilay-PC:~/Documents/cs359/Assignment$ ./a.out
Sequential time = 13.41 sec
parallel time for  2 threads = 6.03 sec and speedup = 2.224819
parallel time for  4 threads = 3.15 sec and speedup = 4.252475
parallel time for  8 threads = 1.98 sec and speedup = 6.781714
parallel time for  16 threads = 1.97 sec and speedup = 6.798573
nilay@Nilay-PC:~/Documents/cs359/Assignment$
```

Speedup Graph:

Cores	1	2	4	8	16
n=500	1	1.58	2.60	3.52	2.53
n=1000	1	2.01	3.64	5.96	5.81
n=2000	1	2.22	4.25	6.78	6.79



Description:

The Algorithm is Basically Floyd Warshall Algorithm.

- As a first step, initialize the solution matrix to be identical to the input graph matrix.
- The solution matrix is then updated by treating each vertex as an intermediate vertex.
- The plan is to select each vertex one at a time and update any shortest paths that use the selected vertex as an intermediate vertex.
- Vertices 0, 1, 2,.., k-1 are already considered when vertex number k is chosen as an intermediate vertex.
- There are two potential outcomes for every pair of source and destination vertices I j), respectively.
- The shortest path from I to j should not include k as an intermediary vertex. We maintain the current dist[i][j] value.

Inference & Parallel Exploitation:

Because we are using OpenMP parallelism can be done by executing the “for” loop with the help of OpenMP directives

But in retrospect it is basically archived by running matrix multiplication in parallel as Floyd Warshal has elements of matrix multiplication

In the Speedup graph we can see that good speedup is achieved until 8 cores and then speedup either drops down or gets plateaued.

This may be due to communication overheads or maybe because the device used to execute the program has 8 Logical processors physically present as Hardware

Note: The programs may give different results depending on which hardware it is run on. It might also depend on the state of the same hardware such as if other programs are running or not or if the hardware is connected to power supply or getting DC from a battery source.