PARALLELIZATION OF GRAPH ALGORITHMS

Yesha Kaniyawala

ABSTRACT

Graphs are used to solve many real-life problems. They are used to represent networks, including paths in a city or telephone network or circuit network. Graphs are also used in social networks like LinkedIn, Facebook. Other than the IT world, graphs have very wide usage in linguistics, chemistry, physics, biology and, of course, mathematics. With such a broad range of applications in place it becomes extremely important that we make them faster, better and more efficient. In this project we are comparing sequential and parallel implementation of Dijkstra's algorithm, probably the most used graph algorithm with different scheduling types and different number of threads. We'll be plotting different comparison graphs to see clearly how much of improvement are we making. We'll also be looking for any discrepancies if there is.

OBJECTIVES

To compare the sequential and parallel implementation of Dijkstra's algorithm and with different number of threads.

To compare the sequential and parallel implementation of Dijkstra's algorithm with different scheduling types viz static and dynamic scheduling.

To plot graph and compare performances visually.

To understand the rate of growth of difference in execution times of both algorithms.

To promote the use of parallel graph algorithms to boost productivity in the areas utilising shortest path finding and similar algorithms.

To promote parallelization of graph algorithms using OpenMP.

INTRODUCTION

In recent times with the advent of internet, use of graph algorithms has seen a giant upsurge. So, it has become imperative to make it faster, better and more efficient. Graphs are used to solve many real-life problems. They are used to represent networks, including paths in a city or telephone network or circuit network. Graphs are also used in social networks like LinkedIn, Facebook. For example, in Facebook, each person is represented with a vertex. Google maps uses graphs for building transportation systems, they are also used in modelling of social networks, Windows file explorer, even the web. Also, it is extremely common problem to find minimum distance between two nodes say in Google Maps, the web or any real-life problem that needs to minimize the cost to reach state B from state A having multiple options at its disposal. Other than the IT world, graphs have very wide usage in linguistics, chemistry, physics, biology and, of course, mathematics. With such a broad range of applications in place it becomes extremely important that we make them faster, better and more efficient.

REQUIREMENTS

SOFTWARE:

Windows, Linux, macOS or any other operating system

C Compiler

OpenMP Library

HARDWARE:

PC with minimum 2GB of RAM

Processor with 4 or more cores

MODULES

Dijkstra's Algorithm (Serial Implementation)

For a given source node in the graph, the algorithm finds the shortest path between that node and every other. It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined. It takes number of vertices, number of edges, the weighted edges and starting edge from where the distances have to be found. Weighted edges are entered in the format u v w where u is the node from where edge starts, v is the destination node and w is the weight of the edge. This algorithm outputs minimum distances to all the edges, infinity, if there is no path connecting u and v and also the time it took for execution.

Dijkstra's Algorithm (Parallel Implementation)

It's the parallel implementation of Dijkstra's algorithm. The input and output format is same for both. Both of them have these functions except for Dijkstra_Parallel(), the former has Dijkstra_Serial() instead. printShortestPathLength() // final output function findEdgeWeight() // to find the edge weight of directly connecting edge, infinity, if not directly connected.

minimumPathLength() // path length to closest unvisited vertex minimumPathVertex()

// closest unvisited vertex

Dijkstra_Parallel() // parallel Dijkstra's implementation

Main() // to get inputs and call Dijkstra's implementation

Random Graph Generator Function

I have created a random graph generating function to generate random graph of desired sizes i.e. of desired number of vertices and edges. Graphs generated from this function will be undirected. There will be no repeating edges and no self-loops. It should be noted that an undirected simple graph can't have more than n*(n-1)/2 edges so we have to set the inputs accordingly but even if you do our function will generate you a random graph with edges less than n*(n-1)/2. Input ranges are given below.

$$2 <= \text{Number of Vertices} <= 10^9$$

$$0 <= \text{Number of edges} <= 10^9$$

$$0 <= \text{Edge\_Weight} <= 10^9$$

IMPLEMENTATION WITH RESULTS AND DESCRIPTIONS

Step 1: Setting the number of vertices, edges we need and the max weight allowed for an edge. This function will generate the exact number of vertices and columns we enter but, in case, if number of edges is greater than the maximum number of allowed edges between n nodes for a simple graph viz. n*(n-1)/2, it will give you random number of edges not exceeding the max edge limit we enter.

```
#include<bits/stdc++.h>
using namespace std;

#define MAX_VERTICES 10
#define MAX_EDGES 30
#define MAXWEIGHT 200

int main()
{
    set<pair<int, int>> container;
    set<pair<int, int>>::iterator it;

    srand(time(NULL));
```

Step 2: Run the input generator function for required number of edges and vertices, copy the output i.e. random edges in u v w format. Save the input in your input file. (“Input.txt” in this example)

Note: Remember to change the name of input file in the given code for both the algorithms.

File Edit Format View Help

```
10 30
0 0 91
0 4 55
0 7 72
0 8 175
0 9 76
1 1 6
3 0 32
3 1 24
3 2 47
```

Step 3: Run the parallel code with the required configuration (set number of threads, scheduling type etc). omp_set_dynamic(0) makes sure that we get the number of threads we

ask for, omp_set_num_threads(**n**) will give you n threads and #pragma omp parallel for schedule(**type**) private(i) can be used to set the scheduling type viz. dynamic or static.

```
omp_set_dynamic(0); // to make sure desired num of threads are alloted
omp_set_num_threads(4); // required number of threads
#pragma omp parallel for schedule(dynamic) private(i)
for(int i = 0; i < V; i++)
{
    if(vertices[i].visited == false)
    {
        int c = findEdgeWeight( u, vertices[i], edges, weights);
        path_length[vertices[i].label] = std::min(path_length[vertices[i].label]
    }
}
```

```
threads = 4

threads = 4

threads = 4

threads = 4

threads = 4

threads = 4

VERTEX  SHORTEST PATH LENGTH
0       0
1       137
2       214
3       Infinity
4       55
5       135
6       271
7       72
8       114
9       76

Running time: 49.999952 ms
```

Step 4: Run codes with different execution configurations and compare the executions times.

Execution Style 1: Serial code

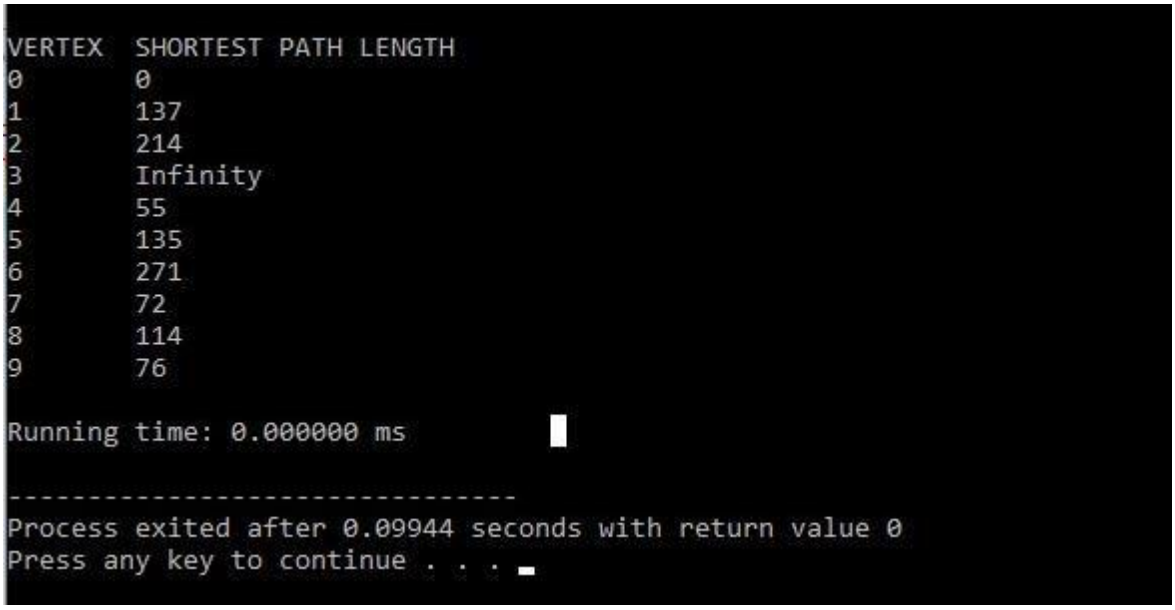
Execution Style 2: Parallel Code (Threads:1, Scheduling: “dynamic”)

Execution Style 3: Parallel Code (Threads:2, Scheduling: “dynamic”)

Execution Style 4: Parallel Code (Threads:3, Scheduling: “dynamic”)

Execution Style 5: Parallel Code (Threads:4, Scheduling: “dynamic”)

Execution Style 6: Parallel Code (Threads:4, Scheduling: “static”)



(output for serial execution parallel execution output will also be similar)

Result Discussion with comparison

Nodes	Edges	Serial	1 thread	2 threads	3 threads	Dynamic (4)	Static (4)
5	10	0.000000	0.000000	0.999928	1.000166	1.999855	1.000166
15	100	0.000000	0.999928	2.000093	3.000021	3.000021	3.000021
50	1000	3.000021	6.000042	6.999969	8.000135	8.000135	6.999969
100	2500	31.999826	36.000013	26.999950	23.999929	20.999908	21.999836
110	5000	69.000006	74.000120	47.999859	39.999962	36.000013	38.000107
150	10000	217.999935	228.000164	131.000042	106.000185	88.000059	96.999884

250	25000	1711.000204	1726.999998	896.000147	687.999964	574.000120	625.000000
500	50000	14792.999983	14794.999838	7510.999918	5736.000061	4815.999985	5039.000034
550	75000	28027.999878	28036.000013	14469.000101	10863.000154	8842.999935	9332.000017
600	100000	42276.999950	42252.999783	21411.999941	16440.999985	13769.000053	14794.999838

* Took average of 5 iterations to get more accurate readings.

Nodes: Number of nodes in the graph.

Edges: Number of edges in the graph. **Serial**:

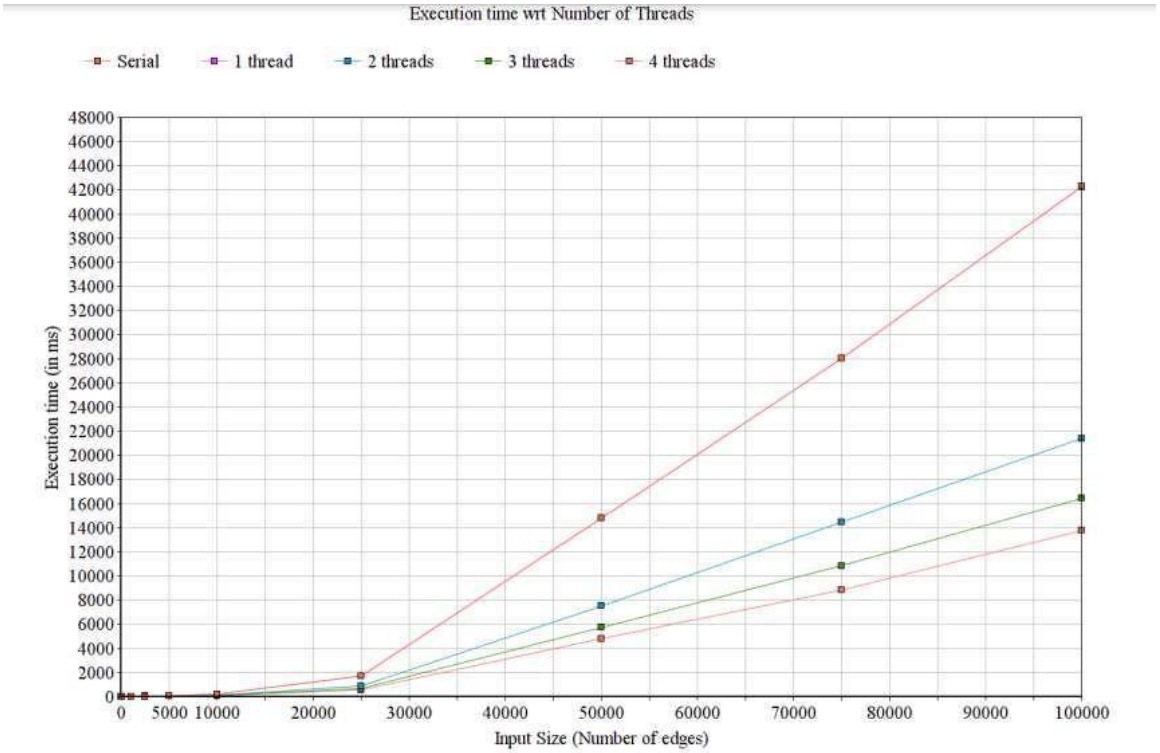
Serial Execution time (in ms).

n threads: Execution time of parallel code with ‘n’ threads (in ms).

Dynamic: Execution time of dynamic scheduling with 4 threads (in ms).

Static: Execution time of Static scheduling with 4 threads(in ms).

EXECUTION TIME WRT NUMBER OF THREADS



Observations:

Serial execution and Parallel code with 1 thread take almost same time, difference is less than half seconds even for 100,000 edges (their graphs are overlapping).

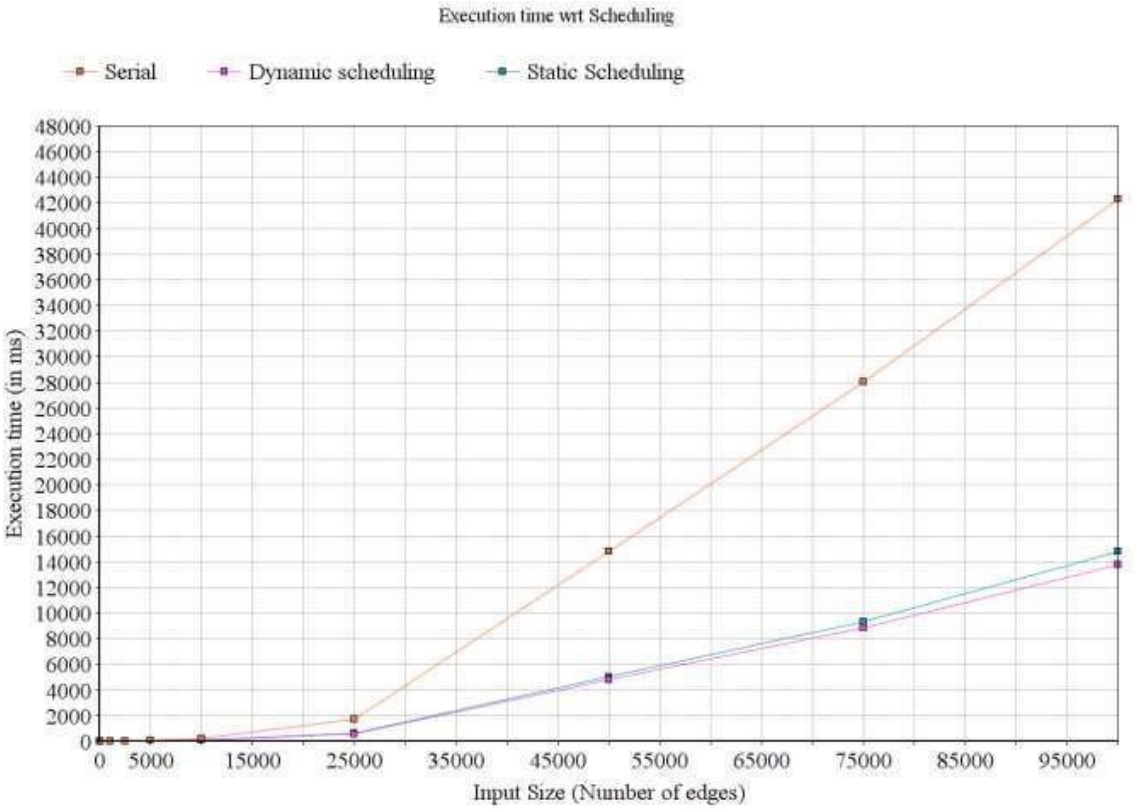
Parallel Code with 2 threads and dynamic scheduling takes a lot less time.

Parallel Code with 3 thread and dynamic scheduling improves the execution time.

Parallel Code with 4 thread and dynamic scheduling further improves it.

Even though there is improvement as we increase the number of edges but the most significant improvement is when me use 2 threads instead of one. The improvement gradually decreases.

EXECUTION TIME WRT SCHEDULING



Observations:

Dynamic scheduling performs slightly better than static scheduling because the loop execution time is not constant and depends on the graph configuration. Static scheduling is better when the loop execution time is constant. Because it assigns equal workload to all threads prior to loop execution so there's no overhead unlike dynamic scheduling but this is not the case here.

CONCLUSION

Parallel execution time is better than serial execution time with the exception when number of threads is equal to 1. In that case it's almost same.

Performance improvement is best when we move from single to double thread, execution time decreases by a factor of 2, then it decreases.

Execution time is best when we use 4 threads with dynamic scheduling.

Dynamic Scheduling is slightly better than Static Scheduling because the time taken to execute the body of loop is not known before.[1]

Even for the same number of vertices and edges the execution times can differ significantly due to graph distribution.

Note: Five threads performed worse than four threads in my quadcore computer due to thread stealing time from thread of the same process. [2]

[1] https://stackoverflow.com/questions/40069587/advantages-and-disadvantages-with-staticand-dynamic-scheduling

[2]: https://stackoverflow.com/questions/3126154/multithreading-what-is-the-point-of-morethreads-than-cores

REFERENCES

Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs,". Numerische Mathematik 1: 269–271. doi:10.1007/BF01386390.

Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "Section
24.3: Dijkstra's algorithm". Introduction to Algorithms (Second ed.). MIT Press and McGrawHill. pp. 595–601. ISBN 0-262-03293-7.

A. Crauser, K. Mehlhorn, U. Meyer, P. Sanders, “A parallelization of Dijikstra’s shortest path algorithm”, in Proc. of MFCS’98, pp. 722-731, 1998.

Y. Tang, Y. Zhang, H. Chen, “A Parallel Shortest Path Algorithm Based on GraphPartitioning and Iterative Correcting”, in Proc. of IEEE HPCC’08, pp. 155-161, 2008.

G. Stefano, A. Petricola, C. Zaroliagis, “On the implementation of parallel shortest path algorithms on a supercomputer”, in Proc. of ISPA’06, pp. 406-417, 2006.

https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-

regionsavailabilityzones.html

http://www.csl.mtu.edu/cs2321/www/newLectures/30_More_Dijkstra.htm

https://www.adamconrad.dev/blog/shortest-paths/

https://medium.com/@farruk/practical-dijkstras-algorithm-b329ade79a1e

https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Ye-Fall-2012-CSE633.pdf

Annexure: Coding

Serial Dijkstra's Algorithm

```
#include <stdio.h>

#include <omp.h>

#include<algorithm>

#define INFINITY 999999


int V,E;


typedef struct

{

int label;

bool visited;


} Vertex;


typedef struct

{

int u;

int v;
```

```
} Edge;


void printShortestPathLength(int *path_length)

{

printf("\nVERTEX \tSHORTEST PATH LENGTH \n");

int i;

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

{

printf("%d \t",i);

if (path_length[i]<INFINITY)

        printf("%d\n",path_length[i]);

else

        printf("Infinity\n");

}

}


int findEdgeWeight(Vertex u, Vertex v, Edge *edges, int *weights)

{


int i;

for(i = 0; i < E; i++)
```

```
{


if(edges[i].u == u.label && edges[i].v == v.label)
{
        return weights[i];
}
}

return INFINITY;
}

int minimimPathLength(int *path_length, Vertex *vertices)
{
int i;
int min_path_length = INFINITY;
for(i = 0; i < V; i++)
{
if(vertices[i].visited == true)
{
        continue;
}
```

```
else if(path_length[i] < min_path_length)
{
        min_path_length = path_length[i];

}

}
return min_path_length;
}


int minimimPathVertex(Vertex *vertices, int *path_length)
{
int i;
int min_path_length = minimimPathLength(path_length, vertices);

for(i = 0; i < V; i++)
{
if(vertices[i].visited == false && path_length[vertices[i].label] == min_path_length)
{
```

```
            vertices[i].visited = true;
            return i;
}
}
}


void Dijkstra_Serial(Vertex *vertices, Edge *edges, int *weights, Vertex *root)
{

double serial_start, serial_end;
int path_length[V];


root->visited = true;
path_length[root->label] = 0;

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

if(vertices[i].label != root->label)
```

```
{
        path_length[vertices[i].label] = findEdgeWeight(*root, vertices[i], edges, weights);
}
}

serial_start = omp_get_wtime();
for(int j = 0; j < V; j++)
{
Vertex u;

int h = minimimPathVertex(vertices, path_length);
u = vertices[h];

for(int i = 0; i < V; i++)
{
        if(vertices[i].visited == false)
        {
                int c = findEdgeWeight( u, vertices[i], edges, weights);
                path_length[vertices[i].label] = std::min(path_length[vertices[i].label],
path_length[u.label] + c);
        }
}
```

```
}

serial_end = omp_get_wtime();

printShortestPathLength(path_length);

printf("\nRunning time: %lf ms\n", (serial_end - serial_start)*1000);

}


int main()

{

freopen("input.txt", "r", stdin);

scanf("%d",&V);

scanf("%d",&E);

Vertex vertices[V];

Edge edges[E];

int weights[E];


int i;

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

{

Vertex a = { .label =i , .visited=false};

vertices[i] = a;

}
```

```
int from,to,weight;

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

{

scanf("%d %d %d",&from,&to,&weight);

Edge e = {.u = from , . v = to};

edges[i] = e;

weights[i] = weight;

}


int source;


scanf("%d",&source);

Vertex root = {source, false};


Dijkstra_Serial(vertices, edges, weights, &root);


return 0;

}
```

Parallel Dijkstra's Algorithm

```
#include <stdio.h>

#include <omp.h>

#include<algorithm>

#define INFINITY 999999


int V,E;


// typedef keyword is used to assign a new name to any existing data-type but it can

// also be used to assign a new name to structure which is a user-defined datatype.

// Syntax: typedef current_name new_name;


typedef struct

{

        int label;

        bool visited;


} Vertex;


typedef struct

{

        int u;

        int v;
```

```
} Edge;


void printShortestPathLength(int *path_length)
{
	printf("\nVERTEX \tSHORTEST PATH LENGTH \n");
	int i;
	for(i = 0; i < V; i++)
	{
		printf("%d \t",i);
		if (path_length[i]<INFINITY)
			printf("%d\n",path_length[i]);
		else
			printf("Infinity\n");
	}
}



//Weight of the edge that connects Vertex u with Vertex v (direct edge)
int findEdgeWeight(Vertex u, Vertex v, Edge *edges, int *weights)
{

	int i;
	for(i = 0; i < E; i++)
	{
```

```
            if(edges[i].u == u.label && edges[i].v == v.label)
            {
                return weights[i];
            }
        }

        return INFINITY;
}

//Gets the minimum path length among all the paths (of unvisited nodes)
int minimimPathLength(int *path_length, Vertex *vertices)
{
        int i;
        int min_path_length = INFINITY;
        for(i = 0; i < V; i++)
        {
            if(vertices[i].visited == true)
            {
                continue;
            }

            else if(path_length[i] < min_path_length)
            {
```

```
                min_path_length = path_length[i];


            }


        }
        return min_path_length;
}



int minimimPathVertex(Vertex *vertices, int *path_length)
{
        int i;
        int min_path_length = minimimPathLength(path_length, vertices);

        // Get the unvisited vertex with the minimum path length
        //Mark it as visited
        for(i = 0; i < V; i++)
        {
            if(vertices[i].visited == false && path_length[vertices[i].label] ==
min_path_length)
            {
                vertices[i].visited = true;
                return i;
            }
```

```
        }
}

// Dijkstra Algorithm
void Dijkstra_Parallel(Vertex *vertices, Edge *edges, int *weights, Vertex *root)
{

        double parallel_start, parallel_end;
        int path_length[V]; // we are creating a vector to store path-length to all the vertices from root node


        root->visited = true; // root node visited
        path_length[root->label] = 0; // path_length to self


        // Compute direct edge weight to all vertices
        for(int i = 0; i < V;i++)
        {

                if(vertices[i].label != root->label)
                {
                        path_length[vertices[i].label] = findEdgeWeight(*root, vertices[i], edges, weights);
                }
```

```
}

parallel_start = omp_get_wtime();
for(int j = 0; j < V; j++)
{
        Vertex u;

        // Obtain the vertex which has shortest distance from root node
        int h = minimimPathVertex(vertices, path_length);
        u = vertices[h];

        // There is something called dynamic teams that could still pick smaller
        // number of threads if the run-time system deems it more appropriate.
        // You can disable dynamic teams by calling omp_set_dynamic(0) or
        // by setting the environment variable

        //   https://stackoverflow.com/questions/11095309/openmp-set-num-threads-is-not-working

        //Update shortest path wrt new source

        omp_set_dynamic(0); // to make sure desired num of threads are alloted
        omp_set_num_threads(4); // required number of threads
        #pragma omp parallel for schedule(static) private(j)
```

```
                for(int i = 0; i < V; i++)
                {
                        if(vertices[i].visited == false)
                        {
                                int c = findEdgeWeight( u, vertices[i], edges, weights);
                                path_length[vertices[i].label] =
std::min(path_length[vertices[i].label], path_length[u.label] + c);
                        }

                }
        }
        //      printf("\nthreads = %d\n", omp_get_num_threads());
        parallel_end = omp_get_wtime();
        printShortestPathLength(path_length);
        printf("\nRunning time: %lf ms\n", (parallel_end - parallel_start)*1000);
}

int main()
{
        freopen("input.txt", "r", stdin);
        scanf("%d",&V);
        scanf("%d",&E);
        Vertex vertices[V]; // Array of struct vertex named vertices[]
        Edge edges[E]; // Array of struct edge named edges[]
```

```
int weights[E]; // contains weights of edge 0,1,2,3.....


// Vertex struct contains label and visit status
// Edge struct contains starting and ending vertex
// labeling starts from zero.


int i;
for(i = 0; i < V; i++)
{
        Vertex a = { .label =i , .visited=false};
        vertices[i] = a;
}



int from,to,weight;
for(i = 0; i < E; i++)
{
        scanf("%d %d %d",&from,&to,&weight);
        Edge e = {.u = from , . v = to};
        edges[i] = e;
        weights[i] = weight;
}


int source;
```

```
    scanf("%d",&source);
    Vertex root = {source, false};

    Dijkstra_Parallel(vertices, edges, weights, &root);

    return 0;
}
```

Random Graph Generator

```
#include <bits/stdc++.h>
using namespace std;

#define MAX_VERTICES 550
#define MAX_EDGES 75000
#define MAXWEIGHT 200

int main()
{
    set<pair<int, int>> container;
    set<pair<int, int>>::iterator it;

    srand(time(NULL));

    int NUM;
    int NUMEDGE;

    ofstream outfile("input.txt"); // Open file for output

    for (int i = 1; i <= 1; i++)
    {
        NUM = MAX_VERTICES;
```

```
NUMEDGE = MAX_EDGES;

while (NUMEDGE > NUM * (NUM - 1) / 2)
    NUMEDGE = 1 + rand() % MAX_EDGES;

outfile << NUM << " " << NUMEDGE << "\n";

for (int j = 1; j <= NUMEDGE; j++)
{
    int a = rand() % NUM;
    int b = rand() % NUM;
    pair<int, int> p = make_pair(a, b);
    pair<int, int> reverse_p = make_pair(b, a);

    while (container.find(p) != container.end() ||
           container.find(reverse_p) != container.end())
    {
        a = rand() % NUM;
        b = rand() % NUM;
        p = make_pair(a, b);
        reverse_p = make_pair(b, a);
    }
```

```
            container.insert(p);
        }

        for (it = container.begin(); it != container.end(); ++it)
        {
            int wt = 1 + rand() % MAXWEIGHT;
            outfile << it->first << " " << it->second << " " << wt << "\n";
        }

        container.clear();
        outfile << "\n";
    }

    outfile << 0 << "\n";
    outfile.close(); // Close file

    return 0;
}
```