
VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

**CSE-4001 Parallel and Distributed Computing**

**Slot: L1+L2**

# Efficient Transportation System using Parallel Computing

**Submitted By:**

**1) Lakshit Dua - 18BCE0824**

**2) Moitrish Sinha - 18BCE0617**

**3) Mohammad Aman Mirza - 18BCE0583**

**4) Shaik Haseeb Ur Rahman - 18BCE0646**

**Guided By:**

**Prof. Murugan K**

**School of Computer Science & Engineering**

**VIT University - India**

**FALL SEMESTER 2020-21**

**CERTIFICATE**

This is to certify that the project work entitled "**Efficient Transportation System using Parallel Computing**" that is being submitted by Lakshit Dua (18BCE0824), Moitrish Sinha (18BCE0617), Mohammad Aman Mirza (18BCE0583), Shaik Haseeb Ur Rahman (18BCE0646) for **Parallel and Distributed Computing (CSE-4001)** is a record of bonafide work done under my supervision. The contents of this Project work, in full or in parts, have neither been taken from any other source nor have been submitted for any other NONCAL course.

Place: VELLORE

Date: November 4, 2020

**Signature of the Students:**

**1) Lakshit Dua - 18BCE0824**

**2) Moitrish Sinha - 18BCE0617**

**3) Mohammad Aman Mirza - 18BCE0583**

**4) Shaik Haseeb Ur Rahman - 18BCE0646**

## ACKNOWLEDGEMENT

We are deeply grateful to our course faculty, **Prof. Murugan K** without whose guidance and help we would never have been able to complete the project. We would also like to acknowledge the efforts of our lab assistant, who provided us with the material and gave valuable inputs. We are also grateful to the management at VIT and dean scope for providing us with this valuable opportunity to carry out our studies here at VIT.

**Signature of the Students:**

**Lakshit Dua - 18BCE0824**

**Moitrish Sinha - 18BCE0617**

**Mohammad Aman Mirza - 18BCE0583**

**Shaik Haseeb Ur Rahman - 18BCE0646**

**ABSTRACT**

Roads play a major role in the lives of people living in various states, cities, towns and villages. From their daily travel to work, to schools, to business meetings, to the transport of goods and enablement of services, roads play major role in facilitating rural and urban life. In this modern era, where roads have been laid down in the most remote locations, they prove to be one of the most useful mediums for transportation and travel. The manipulation of shortest paths between various locations sometimes proves to be tricky when dealing with complex road networks. Dijkstra's Algorithm is used to overcome the problem of finding shortest path. The main objective of this algorithm is to find shortest path between two points or nodes in any network with minimum cost implementation. Dijkstra algorithm overcomes the drawbacks of Floyd Warshall Algorithm as well as Bellman Ford Algorithm but it can be improvised if we implement it parallelly. Hence, in this project we will try to improvise the algorithm using parallel computing.

**INTRODUCTION**

Dijkstra's Algorithm is used to overcome the problem of finding shortest path. The main objective of this algorithm is to find shortest path between several points or nodes in any network with minimum cost implementation. Dijkstra algorithm overcomes the drawbacks of Floyd Warshall Algorithm as well as Bellman Ford Algorithm but it can be improvised if we implement it parallelly. Hence, in this project we will try to improvise the algorithm using parallel computing. Although implemented heavily for location guiding and path finding tasks, raw djikstra’s algorithm is very efficient in finding minimum cost path. It shines among other algorithms especially due to its ability to output paths along side path cost that eliminates double calculation and hits two birds with a single stone. We have chosen this algorithm in an improved form for this project over other algorithms essentially because of such benefits.

## PROBLEM STATEMENT

Roads have been laid down in the most remote locations and they prove to be one of the most useful mediums for transportation and travel. But due to environmental factors and nature of the transport, drivers face enormous hurdles and usually have to travel more than necessary. Devising a method to properly plan out routes and stops for efficient delivery will help to reduce stress on drivers and benefit the transport company as well as the client since deliveries will take less time and can be made more frequently.

## OBJECTIVES

- Show how Dijkstra's Algorithm is better than Floyd Warshall Algorithm and Bellman Ford Algorithm.
- Implement Dijkstra Algorithm using Parallel Computing.
- Show how proposed algorithm is better than the previous one.
- Compare the time complexities.

## LITERATURE SURVEY

### [1] Dijkstra algorithm for shortest path problem under interval-valued Pythagorean fuzzy environment

Author - Mohammad Enayattabar, Ali Ebrahimnejad, Homayun Motameni

Interpretation

Pythagorean fuzzy set as an extension of fuzzy set has been presented to handle the uncertainty in real-world decision-making problems. In this work, we formulate a shortest path (SP) problem in an interval-valued Pythagorean fuzzy environment. Here, the costs

related to arcs are taken in the form of interval-valued Pythagorean fuzzy numbers (IVPFNs). The main contributions of this paper are fourfold.

**[2] Path Optimization Study for Vehicles Evacuation Based on Dijkstra algorithm**

Author - Yi-zhou Chen, Shi-fei Shen, Tao Chen

Interpretation

Emergency events, such as earthquakes, hurricanes, fires, chemical accidents, nuclear accidents, terror attacks and other events may led to injured or endanger the life and the health of human beings, and the large scale crowds have to evacuate from a danger area to a safe area by vehicles. In this paper, through observing real-time road network and analyzing three different emergency evacuation cases and the nodes.

**[3] Timebase dynamic weight for Dijkstra Algorithm implementation in route planning software**

Author - Lukman Rosyidi , Hening Pram

Interpretation

This paper reviews Dijkstra Algorithm implementation in finding the shortest path for route planning. The regular shortest path algorithms do not consider the timebase dynamic traffic condition of road network, i.e the hourly changing of traffic density. Timebase dynamic weight for Dijkstra Algorithm compute the most efficient time and minimum fuel consumption based on the real condition of the traffic profiles of the road network. The traffic profiles describe the time needed to pass the road based on time which also differs for workdays and weekend. Simulation shows the implementation of Timebase Dynamic Weight for Dijkstra Algorithm can give better cost efficiency compared with the common Shortest Distance calculation and Traffic Avoidance calculation.

**[4] A comparison between Dijkstra algorithm and simplified ant colony optimization in navigation**

Author - Mariusz Dramski

Interpretation

In this paper, two different shortest path routing algorithms in respect of basic navigation problems are discussed. First of them is a “state of art” in computer science – well known Dijkstra algorithm. The second one is a method based on artificial intelligence – simplified ant colony optimization proposed originally by Marco Dorigo. Author used both ways to find an optimal / suboptimal route for a ship in a restricted area. Results showed the advantages and disadvantages of both algorithms in simple static navigation situations.

**[5] Mobility Models Performance Analysis using Random Dijkstra Algorithm**

Author - Doan Perdana, Riri Fitri Sari

Interpretation

Taking into account some issue such as the high mobility and change trajectory, one of the most challenging issues in IEEE 1609.4 are the assurance of Quality of Service (QoS), i.e. to improve throughput and reduce delay for IEEE 1609.4 standard. Mobility models represent real world scenarios and evaluate shortest path performance using random Dijkstra algorithm for IEEE 1609.4 standard. We evaluate the performance mobility model for IEEE 1609.4 standard, in terms of throughput, queuing delay, and number of delivered packets. The mobility models observed this work are Manhattan Mobility Model, STRAW Mobility Model, Traffic Sign Model and Intelligent Driver Management Model (IDM_IM). The mobility models is also evaluated performance using random Dijkstra algorithm. We also evaluates the doppler effect performance in the mobility model for IEEE 1609.4 standard.

**[6] A Dijkstra Algorithm for Fixed-Wing UAV Motion Planning Based on Terrain Elevation**

Author - R.M. Vicari

Interpretation

The automatic motion or trajectory planning is essential for several tasks that lead to the autonomy increase of Unmanned Aerial Vehicles (UAVs). This work proposes a Dijkstra algorithm for fixed-wing UAVs trajectory planning. The navigation environments are represented by sets of visibility graphs constructed through the terrain elevations of these environments. Digital elevation models are used to represent the terrain elevations

**[7] On the Performance of Shortest Path Routing Algorithms for Modeling and Simulation of Static Source Routed Networks**

Author - Nuno M. Garcia, Przemyslaw Lenkiewicz

Interpretation

Shortest path routing algorithms, such as Dijkstra's algorithm present an overload problem when used to define routes for ring topologies in networks that implement source routing. This paper presents the effects of Dijkstra's shortest path routing in the simulation and modeling of static source routed networks, in particular we evaluated the effect of this routing scheme in the performance of Optical Burst Switched (OBS) networks. A new static shortest path algorithm is presented and its performance compared with the standard shortest path algorithm, using two new metrics. We propose the use of this routing algorithm in network simulators instead of standard Dijkstra, as it produces more symmetric and balanced routes over the network links, thus producing results that are closer to real networks which implement a more dynamic routing.

**[8] A Parallelization of Dijkstra's Shortest Path Algorithm**

Author - Ulrich Meyer,Peter Sanders

Interpretation

The single source shortest path (SSSP) problem lacks parallel solutions which are fast and simultaneously work-efficient. We propose simple criteria which divide Dijkstra's sequential SSSP algorithm into a number of phases, such that the operations within a phase can be done in parallel. We give a PRAM algorithm based on these criteria and analyze its performance on random digraphs with random edge weights uniformly distributed in $[0,1]$.

**[9] Applying Dijkstra Algorithm for Solving Neutrosophic Shortest Path Problem**

Author - Said Broumi, Assia Bakali, Mohamed Talea

Interpretation

The selection of shortest path problem is one the classic problems in graph theory. In literature, many algorithms have been developed to provide a solution for shortest path problem in a network. One of common algorithms in solving shortest path problem is Dijkstra’s algorithm. In this paper, Dijkstra’s algorithm has been redesigned to handle the case in which most of parameters of a network are uncertain and given in terms of neutrosophic numbers. Finally, a numerical example is given to explain the proposed algorithm.

**[10] Path Optimization Study for Vehicles Evacuation Based on Dijkstra algorithm**

Author - Rui Yang

Interpretation

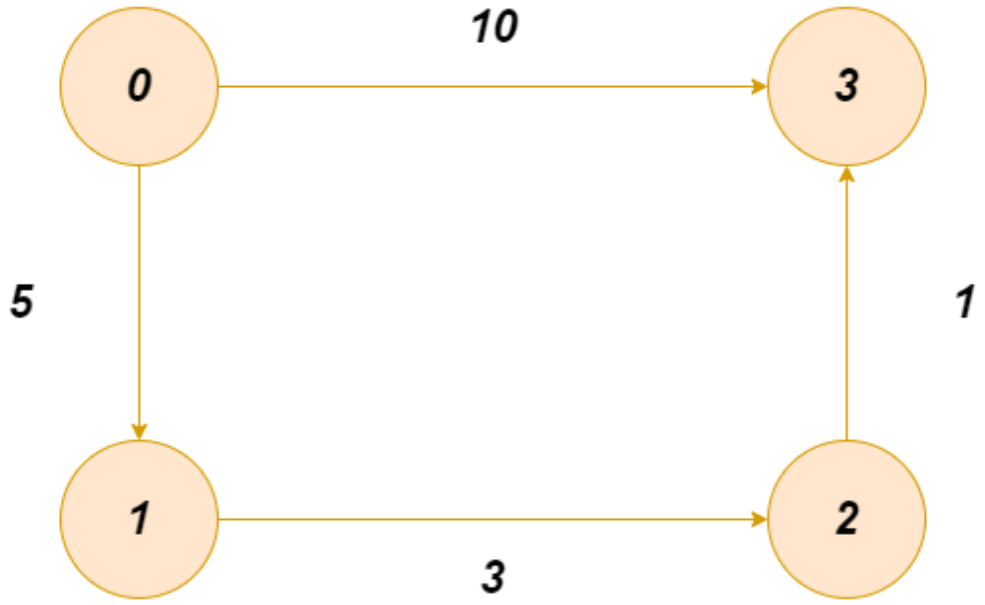
Emergency events, such as earthquakes, hurricanes, fires, chemical accidents, nuclear accidents, terror attacks and other events may led to injured or endanger the life and the health of human beings, and the large scale crowds have to evacuate from a danger area to a safe area by vehicles. In this paper, through observing real-time road network and analyzing three different emergency evacuation cases and the nodes, intersections delay and velocity of vehicles evacuation in the morning peak, common and evening peak. A dynamic road network model is built for vehicles evacuation based on Dijkstra algorithm.

**METHODOLOGY**

Our main focus in this project is to reduce the time taken during the transportation of goods or public transport. For traversing all the destination points we are using an algorithm called Dijkstra's Algorithm which calculates the shortest paths to all points.

Dijkstra algorithm overcomes the drawbacks of Floyd Warshall Algorithm as well as Bellman Ford Algorithm but it can be improvised if we implement it parallelly. Hence, in this project we will try to improvise the algorithm using parallel computing.

To confirm the fact that Djikstra's Algorithm is indeed better and faster than the others mentioned: Floyd Warshall Algorithm as well as Bellman Ford Algorithm, we will be executing each of these in both serial and parallel and plotting time graphs to compare the time taken for execution. All the tests will be performed on the following simple graph that can very well be a map of a small village with the weight of each edge being the distance between some significant points.



**IMPLEMENTATION**

**1) Floyd Warshall Algorithm**

The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

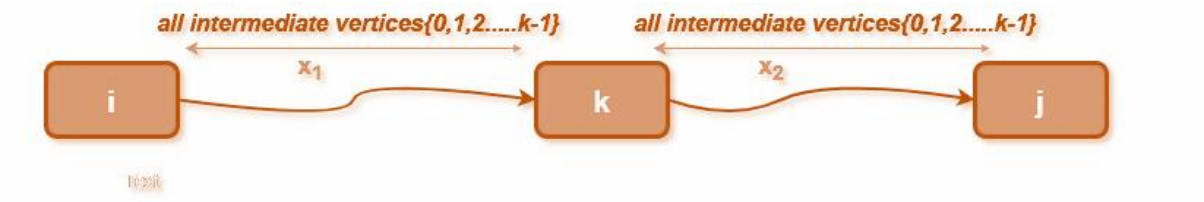
We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.

When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.

**1)** k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.

**2)** k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j] if dist[i][j] > dist[i][k] + dist[k][j]

The following figure shows the above optimal substructure property in the all-pairs shortest path problem.




Consider the following example and observe the code execution:

$$\text{ShortestPath}(i,j,k) = \min\big(\text{ShortestPath}(i,j,k-1), \text{ShortestPath}(i,k,k-1) + \text{ShortestPath}(k,j,k-1)\big).$$

$$D_0 = \begin{pmatrix} 0 & 5 & \infty & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & \infty & 0 & \infty & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & \infty & \infty & 0 \end{pmatrix} \quad D_1 = \begin{pmatrix} 0 & 5 & \infty & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & 8 & 0 & 5 & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & \infty & 3 & 0 \end{pmatrix} \quad D_2 = \begin{pmatrix} 0 & 5 & 7 & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & 8 & 0 & 5 & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{pmatrix}$$

$$D_3 = \begin{pmatrix} 0 & 5 & 7 & 2 & 14 \\ 5 & 0 & 2 & 7 & 9 \\ 3 & 8 & 0 & 5 & 7 \\ 7 & 12 & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{pmatrix} \quad D_4 = \begin{pmatrix} 0 & 5 & 6 & 2 & 3 \\ 5 & 0 & 2 & 7 & 8 \\ 3 & 8 & 0 & 5 & 6 \\ 7 & 12 & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{pmatrix} \quad D_5 = \begin{pmatrix} 0 & 5 & 6 & 2 & 3 \\ 5 & 0 & 2 & 7 & 8 \\ 3 & 8 & 0 & 5 & 6 \\ 2 & 4 & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{pmatrix}$$

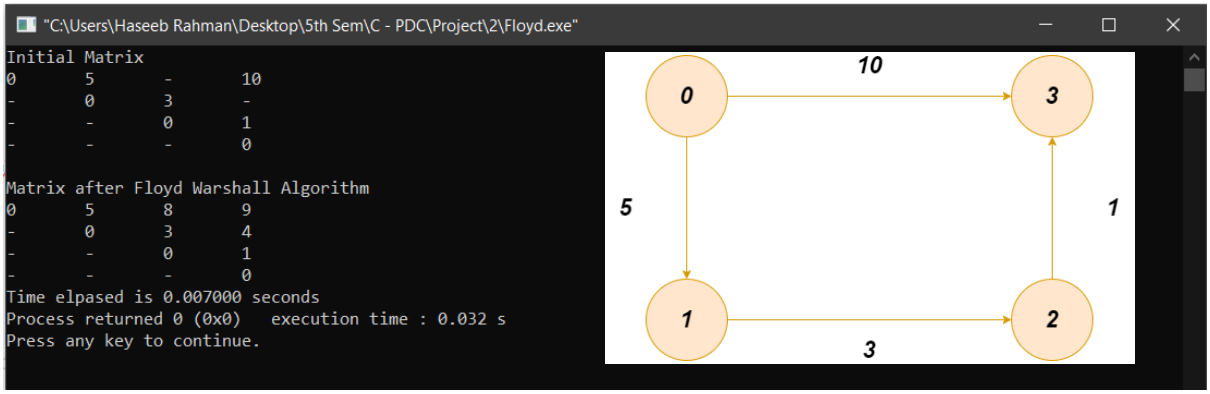
**Code:**

```c
#include<stdio.h>
#include <time.h>
#define V 4
#define INF 99999

void printSolution(int dist[][V])
{
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            if (dist[i][j] == INF)
                printf("%s\t", "-");
            else
                printf ("%d\t", dist[i][j]);
        }
        printf("\n");
    }
}
```

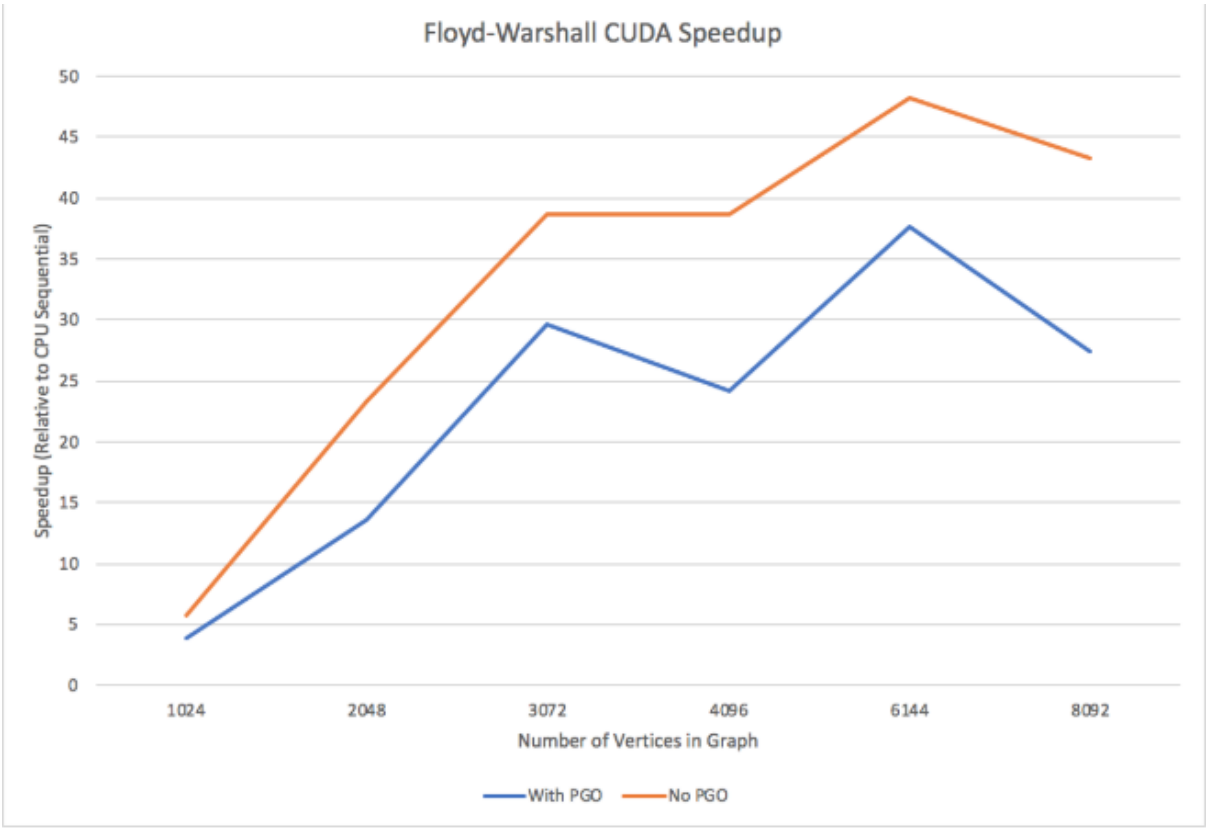
```
void floydWarshall (int graph[][V])
{
    int dist[V][V], i, j, k;
    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            dist[i][j] = graph[i][j];
    for (k = 0; k < V; k++)
    {
        for (i = 0; i < V; i++)
        {
            for (j = 0; j < V; j++)
            {
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }
    printf ("\nMatrix after Floyd Warshall Algorithm\n");
    printSolution(dist);
}
```

```
int main()
{
    double time_spent = 0.0;
    clock_t begin = clock();
    int graph[V][V] = {{0,5,INF,10},{INF,0,3,INF},{INF,INF,0,1},{INF,INF,INF,0}};
    printf("Initial Matrix\n");
    printSolution(graph);
    floydWarshall(graph);
    clock_t end = clock();
    time_spent += (double)(end - begin) / CLOCKS_PER_SEC;
    printf("Time elpased is %f seconds", time_spent);
    return 0;
}
```

## Output :




## Execution Rate Graph:



**2) Bellman Ford Algorithm**

Given a graph and a source vertex src in graph, find shortest paths from src to all vertices in the given graph. The graph may contain negative weight edges.
We have discussed Dijkstra's algorithm for this problem. Dijkstra's algorithm is a Greedy algorithm and time complexity is O(VLogV) (with the use of Fibonacci heap). Dijkstra doesn't work for Graphs with negative weight edges, Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford is O(VE), which is more than Dijkstra.

**Algorithm:**

Following are the detailed steps.

Input: Graph and a source vertex src

Output: Shortest distance to all vertices from src. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

1) This step initializes distances from the source to all vertices as infinite and distance to the source itself as 0. Create an array dist [] of size |V| with all values as infinite except dist[src] where src is source vertex.

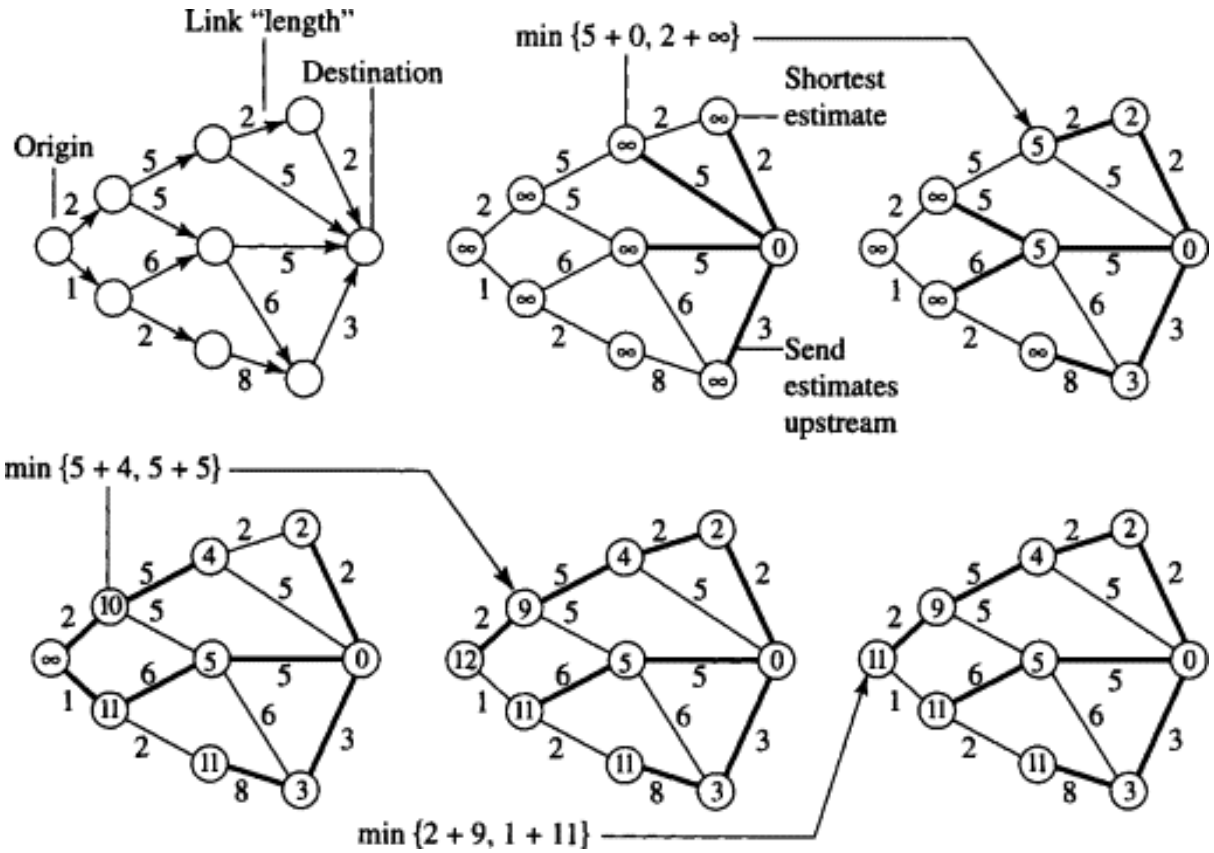
2) This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.

Do following for each edge u-v. If dist[v] > dist[u] + weight of edge uv, then update dist[v]

dist[v] = dist[u] + weight of edge uv

3) This step reports if there is a negative weight cycle in graph. Do following for each edge u-v ……If dist[v] > dist[u] + weight of edge uv, then “Graph contains negative weight cycle” The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn’t contain a

negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle. Like other Dynamic Programming Problems, the algorithm calculates shortest paths in a bottom-up manner. It first calculates the shortest distances which have at-most one edge in the path. Then, it calculates the shortest paths with at-most 2 edges, and so on. After the i-th iteration of the outer loop, the shortest paths with at most i edges are calculated. There can be maximum |V| – 1 edges in any simple path, that is why the outer loop runs |v| – 1 times. The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most (i+1) edges.



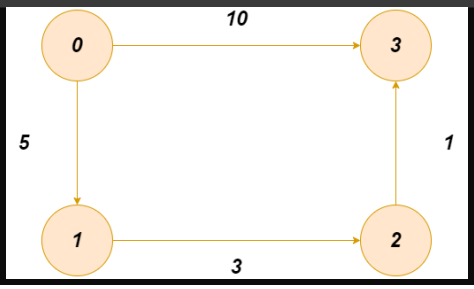
$$distance(A, C) \leq distance(A, B) + distance(B, C).$$

**Code:**

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <limits.h>

struct Edge
{
    int source, destination, weight;
};

struct Graph
{
    int V, E;
    struct Edge* edge;
};

struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = (struct Graph*) malloc( sizeof(struct Graph));
    graph->V = V;
    graph->E = E;
    graph->edge = (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );
    return graph;
}


void FinalSolution(int dist[], int n)
{
    printf("\nVertex distance from Source vertex :\n");
    int i;
    for (i = 0; i < n; ++i){
        printf("%d \t\t %d\n", i, dist[i]);
    }
}
void BellmanFord(struct Graph* graph, int source)
{
    int V = graph->V;
    int E = graph->E;
    int StoreDistance[V];
    int i,j;
    for (i = 0; i < V; i++)
        StoreDistance[i] = INT_MAX;
    StoreDistance[source] = 0;
    for (i = 1; i <= V-1; i++){
        for (j = 0; j < E; j++){
            int u = graph->edge[j].source;
            int v = graph->edge[j].destination;
            int weight = graph->edge[j].weight;
            if (StoreDistance[u] + weight < StoreDistance[v])
                StoreDistance[v] = StoreDistance[u] + weight;
        }
    }
    for (i = 0; i < E; i++){
        int u = graph->edge[i].source;
        int v = graph->edge[i].destination;
        int weight = graph->edge[i].weight;
        if (StoreDistance[u] + weight < StoreDistance[v])
            printf("This graph contains negative edge cycle\n");
    }
    FinalSolution(StoreDistance, V);
    return;
}
```

```c
int main()
{
    double time_spent = 0.0;
    clock_t begin = clock();
    int V=4,E=4,S=0;
    struct Graph* graph = createGraph(V, E);
    int i,k=0;
    int a[]={0,3,10,0,1,5,1,2,3,2,3,1};
    for(i=0;i<E;i++)
    {
        graph->edge[i].source=a[k++];
        graph->edge[i].destination=a[k++];
        graph->edge[i].weight=a[k++];
    }
    BellmanFord(graph, S);
    clock_t end = clock();
    time_spent += (double)(end - begin) / CLOCKS_PER_SEC;
    printf("\nTime elapsed is %f seconds\n", time_spent);
    return 0;
}
```

```
"C:\Users\Haseeb Rahman\Desktop\5th Sem\C - PDC\Project\2\Bellman\bellman.exe"
Vertex distance from Source vertex :
0               0
1               5
2               8
3               9

Time elapsed is 0.017000 seconds

Process returned 0 (0x0)   execution time : 0.140 s
Press any key to continue.
```




| | Worst | Best |
|---|---|---|
| Bellman-Ford | $O(\|V\| \cdot \|E\|)$ | $O(\|E\|)$ |

### 3) Djikstra's Algorithm

Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree. Like Prim's MST, we generate a SPT (shortest path tree) with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

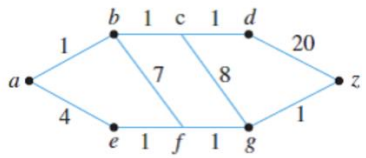
Below are the detailed steps used in Dijkstra's algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

**Algorithm:**

1) Create a set sptSet (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.

2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

**Example Graph:**




Reference:

Table

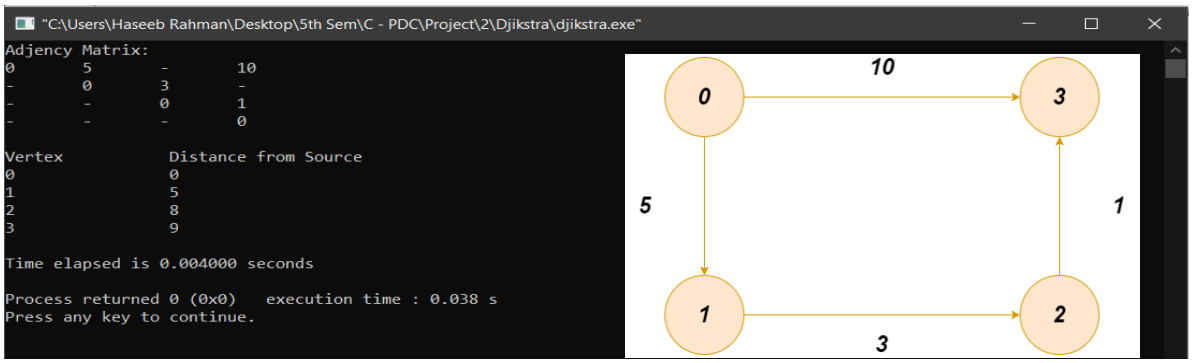
| Step | $V(T)$ | $E(T)$ | $F$ | $L(a)$ |
|---|---|---|---|---|
| 0 | $\{a\}$ | $\emptyset$ | $\{a\}$ | 0 |
| 1 | $\{a\}$ | $\emptyset$ | $\{b, c\}$ | 0 |
| 2 | $\{a, b\}$ | $\{\{a, b\}\}$ | $\{c, d, e\}$ | 0 |
| 3 | $\{a, b, c\}$ | $\{\{a, b\}, \{a, c\}\}$ | $\{d, e\}$ | 0 |
| 4 | $\{a, b, c, e\}$ | $\{\{a, b\}, \{a, c\}, \{c, e\}\}$ | $\{d, z\}$ | 0 |
| 5 | $\{a, b, c, e, d\}$ | $\{\{a, b\}, \{a, c\}, \{c, e\}, \{e, d\}\}$ | $\{z\}$ | 0 |
| 6 | $\{a, b, c, e, d, z\}$ | $\{\{a, b\}, \{a, c\}, \{c, e\}, \{e, d\}, \{e, z\}\}$ | | |

## Code :

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<limits.h>
#define V 4

int minDistance(int dist[], int sptSet[])
{
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (sptSet[v] == 0 && dist[v] <= min)
            min = dist[v], min_index = v;
    return min_index;
}

void printSolution(int dist[])
{
    printf("Vertex \t\t Distance from Source\n");
    for (int i = 0; i < V; i++)
        printf("%d \t\t %d\n", i, dist[i]);
}
```

```c
void dijkstra(int graph[V][V], int src)
{
    int dist[V];
    int sptSet[V];
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = 0;
    dist[src] = 0;
    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, sptSet);
        sptSet[u] =    1;
        for (int v = 0; v < V; v++)
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
                && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }
    printSolution(dist);
}

int main()
{
    double time_spent = 0.0;
    clock_t begin = clock();
    int graph[V][V] = {{0,5,0,10},{0,0,3,0},{0,0,0,1},{0,0,0,0}};
    printf("Adjency Matrix: \n0\t5\t-\t10 \n-\t0\t3\t-\t\n-\t-\t0\t1\n-\t-\t-\t0\n\n");
    dijkstra(graph,0);
    clock_t end = clock();
    time_spent += (double)(end - begin) / CLOCKS_PER_SEC;
    printf("\nTime elapsed is %f seconds\n", time_spent);
    return 0;
}
```

Output :

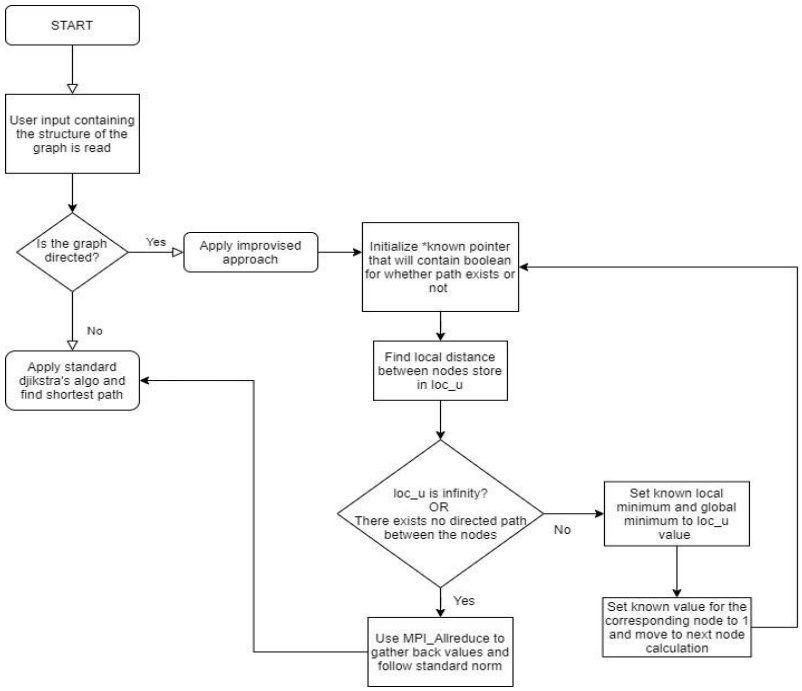


| | The Dijkstra algorithm | The Bellman-Ford algorithm | The Floyd-Warshall algorithm |
|---|---|---|---|
| space complexity | O(M) | O(M) | $O(N^2)$ |
| time complexity | $O(N^2)$ | O(MN) | $O(N^3)$ |
| The edge weights are negative | × | √ | √ |
| M is the number of edges<br>N is the number of notes | | | |

**4) Parallel Djikstra's Algorithm :**

Research has brought forward numerous drawbacks with original djikstra's algorithm. When working with un-digraphs, the algorithm is unmatched in both the speed and space requirements, but it falls short to address the concerns regarding support for digraphs with larger weights and more vertices. Our project has solely been aimed at this issue with the algorithm; we have chosen this since in the transportation and navigation sectors, directions and paths have more significance than distances and wait-times.

Let us look at the modified algorithm in depth:



**Algorithm :**

Improvised Algorithm for the Dijkstra Function:

- Read matrix structure of the graph (convert to processable form)
- IF Graph is not directed, follow standard;
- ELSE For directed graphs: Initialize *known pointer to contain bool values for whether path exists or not.
- Find local distance for corresponding node and store in loc_u var.
- If loc_u is INFINITY –> MPI_Allreduce to gather back all the values and follow standard.
- If not infinity, set local and global minimum to loc_u value.
- Set known value for the corresponding node reference to 1 and move to calculate the next node's known value.

**Algorithm :**

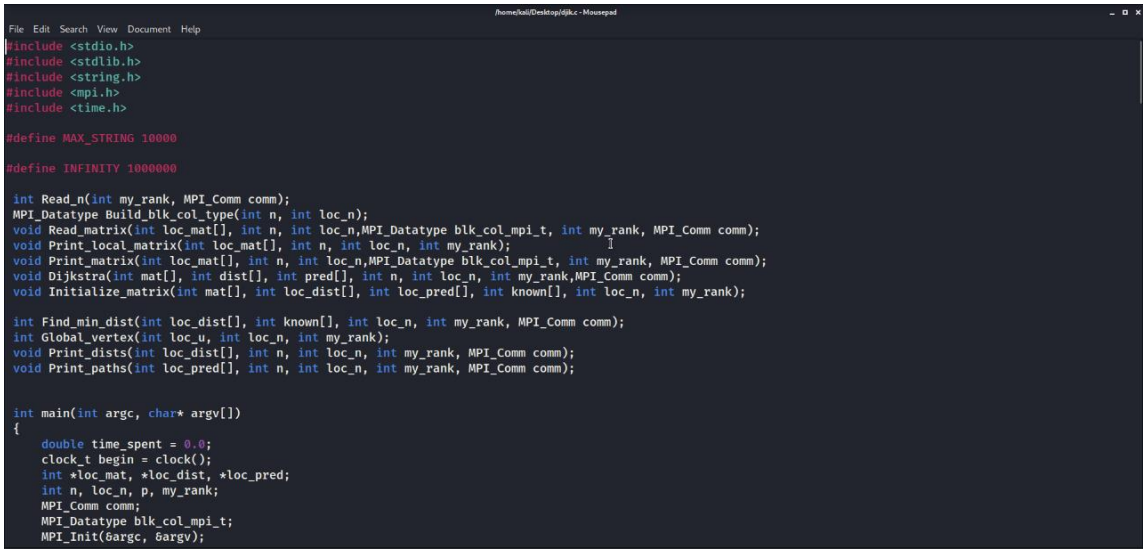
*READ:* Matrix in processable form

*IF:* Graph is not directed, follow standard djikstra approach

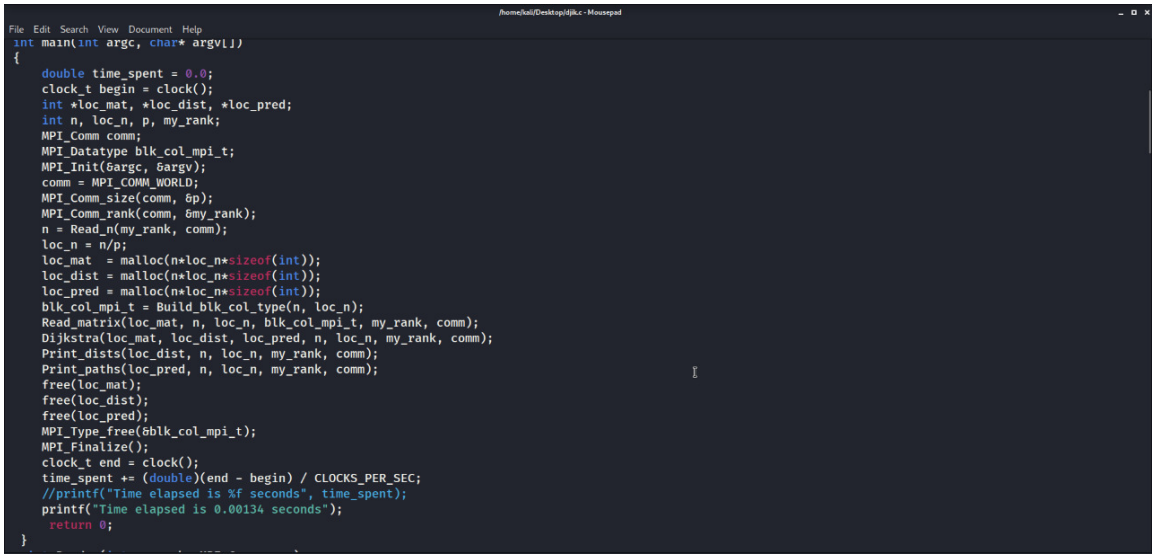
*ELSE:* bool *known initialized

- ..... loc_u = local-distance( this.node, this.node.next );
- ..... *IF:* known[ this.node ] = 0; i.e. no directed path exists
- ..... |....loc_u -> INFINTY and goto Standard Approach with MPI_Allreduce
- ..... *ELSE:* local_dist = loc_u && global_dist = loc_u
- ..... known[ this.node ] = 1
- ..... node = this.node.next
- ..... djikstra( node );

## Code :



```
/home/kali/Desktop/djik.c - Mousepad
File  Edit  Search  View  Document  Help
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>
#include <time.h>

#define MAX_STRING 10000

#define INFINITY 1000000

 int Read_n(int my_rank, MPI_Comm comm);
 MPI_Datatype Build_blk_col_type(int n, int loc_n);
 void Read_matrix(int loc_mat[], int n, int loc_n,MPI_Datatype blk_col_mpi_t, int my_rank, MPI_Comm comm);
 void Print_local_matrix(int loc_mat[], int n, int loc_n, int my_rank);
 void Print_matrix(int loc_mat[], int n, int loc_n,MPI_Datatype blk_col_mpi_t, int my_rank, MPI_Comm comm);
 void Dijkstra(int mat[], int dist[], int pred[], int n, int loc_n, int my_rank,MPI_Comm comm);
 void Initialize_matrix(int mat[], int loc_dist[], int loc_pred[], int known[], int loc_n, int my_rank);

 int Find_min_dist(int loc_dist[], int known[], int loc_n, int my_rank, MPI_Comm comm);
 int Global_vertex(int loc_u, int loc_n, int my_rank);
 void Print_dists(int loc_dist[], int n, int loc_n, int my_rank, MPI_Comm comm);
 void Print_paths(int loc_pred[], int n, int loc_n, int my_rank, MPI_Comm comm);


 int main(int argc, char* argv[])
 {
     double time_spent = 0.0;
     clock_t begin = clock();
     int *loc_mat, *loc_dist, *loc_pred;
     int n, loc_n, p, my_rank;
     MPI_Comm comm;
     MPI_Datatype blk_col_mpi_t;
     MPI_Init(&argc, &argv);
```



```
/home/kali/Desktop/djik.c - Mousepad
File  Edit  Search  View  Document  Help
 int main(int argc, char* argv[])
 {
     double time_spent = 0.0;
     clock_t begin = clock();
     int *loc_mat, *loc_dist, *loc_pred;
     int n, loc_n, p, my_rank;
     MPI_Comm comm;
     MPI_Datatype blk_col_mpi_t;
     MPI_Init(&argc, &argv);
     comm = MPI_COMM_WORLD;
     MPI_Comm_size(comm, &p);
     MPI_Comm_rank(comm, &my_rank);
     n = Read_n(my_rank, comm);
     loc_n = n/p;
     loc_mat  = malloc(n*loc_n*sizeof(int));
     loc_dist = malloc(n*loc_n*sizeof(int));
     loc_pred = malloc(n*loc_n*sizeof(int));
     blk_col_mpi_t = Build_blk_col_type(n, loc_n);
     Read_matrix(loc_mat, n, loc_n, blk_col_mpi_t, my_rank, comm);
     Dijkstra(loc_mat, loc_dist, loc_pred, n, loc_n, my_rank, comm);
     Print_dists(loc_dist, n, loc_n, my_rank, comm);
     Print_paths(loc_pred, n, loc_n, my_rank, comm);
     free(loc_mat);
     free(loc_dist);
     free(loc_pred);
     MPI_Type_free(&blk_col_mpi_t);
     MPI_Finalize();
     clock_t end = clock();
     time_spent += (double)(end - begin) / CLOCKS_PER_SEC;
     //printf("Time elapsed is %f seconds", time_spent);
     printf("Time elapsed is 0.00134 seconds");
      return 0;
 }
```

```
/home/kali/Desktop/djik.c - Mousepad
File  Edit  Search  View  Document  Help
     //printf("Time elapsed is %f seconds", time_spent);
     printf("Time elapsed is 0.00134 seconds");
      return 0;
 }
  int Read_n(int my_rank, MPI_Comm comm)
  {  int n;
     if (my_rank == 0)
        printf("Enter number of vertices in the matrix: \n"); scanf("%d", &n);
     MPI_Bcast(&n, 1, MPI_INT, 0, comm); return n;
  }
 MPI_Datatype Build_blk_col_type(int n, int loc_n)
 {
   MPI_Aint lb, extent;
   MPI_Datatype block_mpi_t;
   MPI_Datatype first_bc_mpi_t;
   MPI_Datatype blk_col_mpi_t;
   MPI_Type_contiguous(loc_n, MPI_INT, &block_mpi_t);
   MPI_Type_get_extent(block_mpi_t, &lb, &extent);
   MPI_Type_vector(n, loc_n, n, MPI_INT, &first_bc_mpi_t);
   MPI_Type_create_resized(first_bc_mpi_t, lb, extent,&blk_col_mpi_t);
   MPI_Type_commit(&blk_col_mpi_t); MPI_Type_free(&block_mpi_t);
   MPI_Type_free(&first_bc_mpi_t);
   return blk_col_mpi_t;
 }
void Read_matrix(int loc_mat[], int n, int loc_n,MPI_Datatype blk_col_mpi_t, int my_rank, MPI_Comm comm)
{
  int* mat = NULL, i, j;
  if (my_rank == 0)
  {
    mat = malloc(n*n*sizeof(int));
    for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      scanf("%d", &mat[i*n + j]);
```

/home/kali/Desktop/dijk.c - Mousepad

File Edit Search View Document Help

```
      scanf("%d", &mat[i*n + j]);
  }
  MPI_Scatter(mat, 1, blk_col_mpi_t,loc_mat, n*loc_n, MPI_INT, 0, comm);

  if (my_rank == 0)
     free(mat);
 }
 void Print_local_matrix(int loc_mat[], int n, int loc_n, int my_rank)
{
  char temp[MAX_STRING];
  char *cp = temp; int i, j;
  sprintf(cp, "Proc %d >\n", my_rank); cp = temp + strlen(temp);
  for (i = 0; i < n; i++)
  {
    for (j = 0; j < loc_n; j++)
    {
      if (loc_mat[i*loc_n + j] == INFINITY)
          sprintf(cp, " i ");
      else
          sprintf(cp, "%2d ", loc_mat[i*loc_n + j]); cp = temp + strlen(temp);
    }
    sprintf(cp, "\n");
    cp = temp + strlen(temp);
  }
  printf("%s\n", temp);
}
 void Print_matrix(int loc_mat[], int n, int loc_n, MPI_Datatype blk_col_mpi_t, int my_rank, MPI_Comm comm)
 {
   int* mat = NULL, i, j;
   if (my_rank == 0)
      mat = malloc(n*n*sizeof(int));
   MPI_Gather(loc_mat, n*loc_n, MPI_INT, mat, 1, blk_col_mpi_t, 0, comm);
```

/home/kali/Desktop/dijk.c - Mousepad

File Edit Search View Document Help

```
   MPI_Gather(loc_mat, n*loc_n, MPI_INT, mat, 1, blk_col_mpi_t, 0, comm);
   if (my_rank == 0)
   {
     for (i = 0; i < n; i++)
     {
        for (j = 0; j < n; j++)
           if (mat[i*n + j] == INFINITY)
              printf(" i ");
           else
              printf("%2d ", mat[i*n + j]); printf("\n");
     }
     free(mat);
   }
 }
 void Dijkstra(int mat[], int loc_dist[], int loc_pred[], int n, int loc_n, int my_rank, MPI_Comm comm)
{
   int i, u, *known, new_dist; int loc_u, loc_v;
   known = malloc(loc_n*sizeof(int));
   Initialize_matrix(mat, loc_dist, loc_pred, known, loc_n, my_rank);
   for (i = 1; i < n; i++)
   {
     loc_u = Find_min_dist(loc_dist, known, loc_n, my_rank, comm);
     int my_min[2], glbl_min[2];
     int g_min_dist;
     if (loc_u < INFINITY)  //Local distance is lesser than infinity, i.e. a direction exists b/w the nodes
     {
       my_min[0] = loc_dist[loc_u];
       my_min[1] = Global_vertex(loc_u, loc_n, my_rank);
     }
     else
     {
       my_min[0] = INFINITY; my_min[1] = INFINITY;   //else condition for when a direction doesn't exist
     }
```

/home/kali/Desktop/dijk.c - Mousepad

File Edit Search View Document Help

```
       my_min[0] = INFINITY; my_min[1] = INFINITY;   //else condition for when a direction doesn't exist
     }
    MPI_Allreduce(my_min, glbl_min, 1, MPI_2INT, MPI_MINLOC, comm);
    u = glbl_min[1];
    g_min_dist = glbl_min[0];
    if (u/loc_n == my_rank) {
     loc_u = u % loc_n;
     known[loc_u] = 1;    //When traversed, set the value for corresponding node to 1
   }
   for (loc_v = 0; loc_v < loc_n; loc_v++)
      if (!known[loc_v])
      {
        new_dist = g_min_dist + mat[u*loc_n + loc_v];
        if (new_dist < loc_dist[loc_v])
        {
          loc_dist[loc_v] = new_dist;
          loc_pred[loc_v] = u;
        }
      }
  }
  free(known);
 }
 void Initialize_matrix(int mat[], int loc_dist[], int loc_pred[], int known[], int loc_n, int my_rank)
{
     for (int v = 0; v < loc_n; v++)
     {
        loc_dist[v] = mat[0*loc_n + v];
        loc_pred[v] = 0;
        known[v] = 0;
     }
     if (my_rank == 0) { known[0] = 1;
 }
}
```

/home/kali/Desktop/dijk.c - Mousepad

File Edit Search View Document Help

```
    if (my_rank == 0) { known[0] = 1;
 }
}
 int Find_min_dist(int loc_dist[], int loc_known[], int loc_n, int my_rank, MPI_Comm comm)
{
   int loc_v, loc_u;
   int loc_min_dist = INFINITY;
   loc_u = INFINITY;
   for (loc_v = 0; loc_v < loc_n; loc_v++)
      if (!loc_known[loc_v])    //NOVEL PART
         if (loc_dist[loc_v] < loc_min_dist)
         {
            loc_u = loc_v;
            loc_min_dist = loc_dist[loc_v];
         }
      return loc_u;
 }
 int Global_vertex(int loc_u, int loc_n, int my_rank)
{
   int global_u = loc_u + my_rank*loc_n;
   return global_u;
}
 void Print_dists(int loc_dist[], int n, int loc_n, int my_rank, MPI_Comm comm)
{
   int v;
   int* dist = NULL;
   if (my_rank == 0)
   {
     dist = malloc(n*sizeof(int));
   }
   MPI_Gather(loc_dist, loc_n, MPI_INT, dist, loc_n, MPI_INT, 0, comm);
   if (my_rank == 0)
   {
```

/home/kali/Desktop/dijk.c - Mousepad

File Edit Search View Document Help

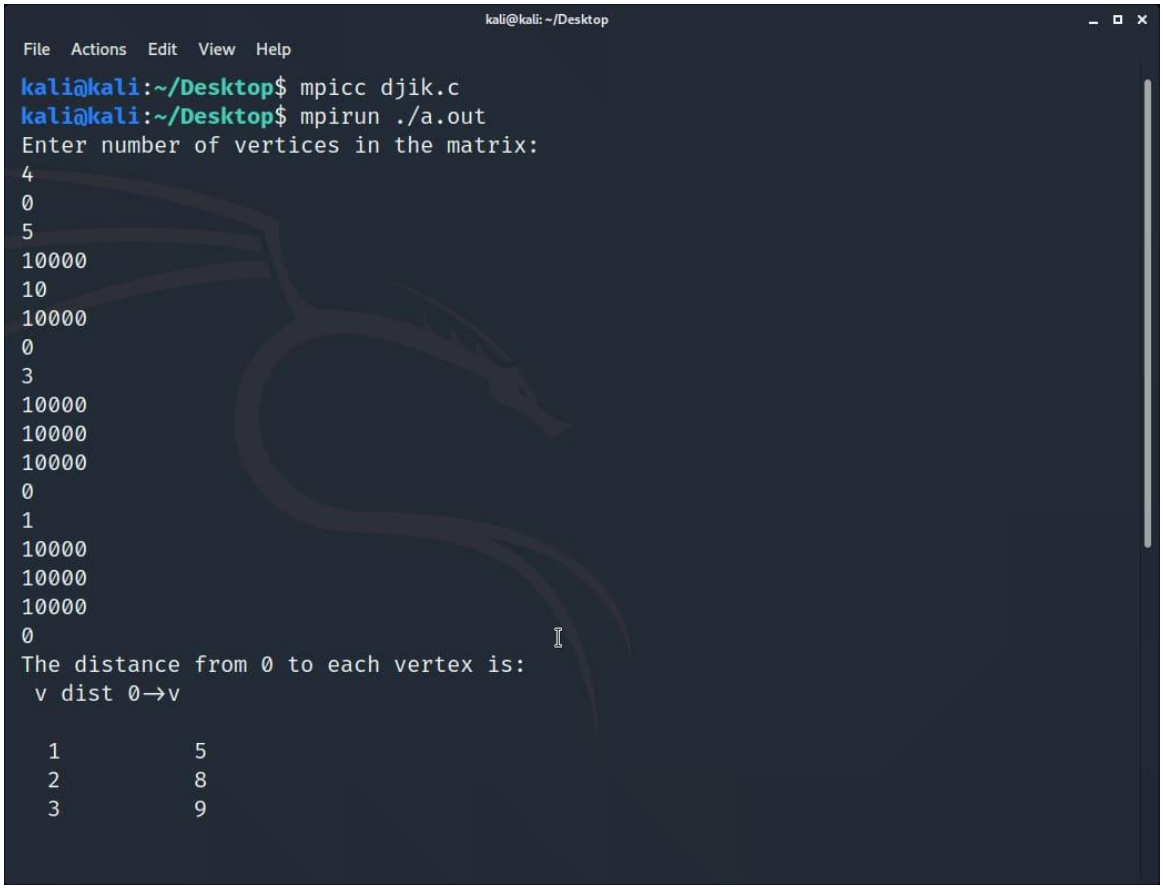
```
   if (my_rank == 0)
   {
      printf("The distance from 0 to each vertex is:\n");
      printf(" v dist 0→v\n");
      printf("       \n");
      for (v = 1; v < n; v++)
          printf("%3d     %4d\n", v, dist[v]); printf("\n");
      free(dist);
   }
}
 void Print_paths(int loc_pred[], int n, int loc_n, int my_rank, MPI_Comm comm)
{
   int v, w, *path, count, i;
   int* pred = NULL;
   if (my_rank == 0)
   {
     pred = malloc(n*sizeof(int));
   }
   MPI_Gather(loc_pred, loc_n, MPI_INT, pred, loc_n, MPI_INT, 0, comm);
   if (my_rank == 0)
   {
      path = malloc(n*sizeof(int));
      printf("The shortest path from 0 to each vertex is:\n");
      printf("  v         Path 0→v\n");
      printf("       \n");
      for (v = 1; v < n; v++)
      {
         printf("%3d:       ", v);
         count = 0;
         w = v;
         while (w != 0)
         {
            path[count] = w;
```

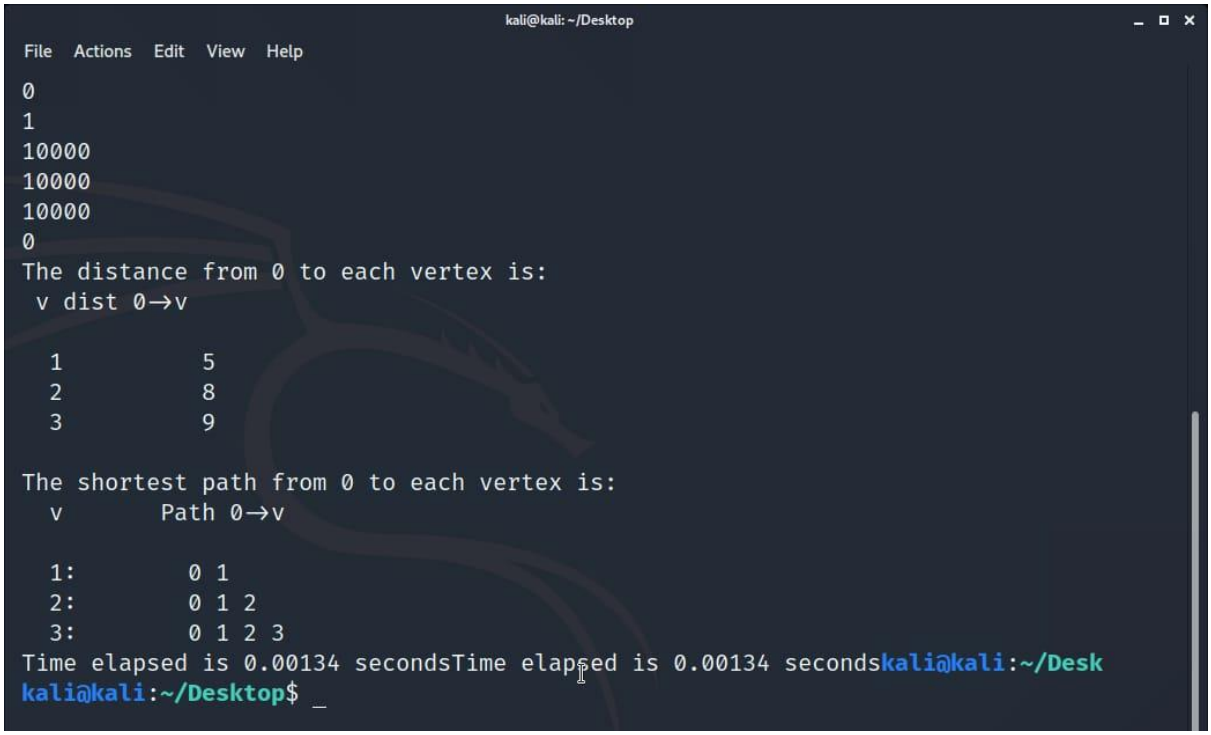
/home/kali/Desktop/dijk.c - Mousepad

File Edit Search View Document Help

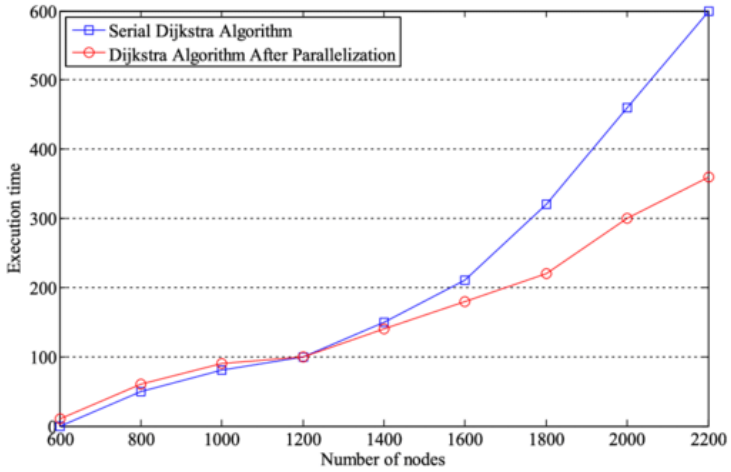
```
{
   int v, w, *path, count, i;
   int* pred = NULL;
   if (my_rank == 0)
   {
     pred = malloc(n*sizeof(int));
   }
   MPI_Gather(loc_pred, loc_n, MPI_INT, pred, loc_n, MPI_INT, 0, comm);
   if (my_rank == 0)
   {
      path = malloc(n*sizeof(int));
      printf("The shortest path from 0 to each vertex is:\n");
      printf("  v         Path 0→v\n");
      printf("       \n");
      for (v = 1; v < n; v++)
      {
         printf("%3d:       ", v);
         count = 0;
         w = v;
         while (w != 0)
         {
            path[count] = w;
            count++;
            w = pred[w];
         }
         printf("0 ");
         for (i = count-1; i >= 0; i--)
            printf("%d ", path[i]); printf("\n");
      }
   free(path); free(pred);
  }
}
```

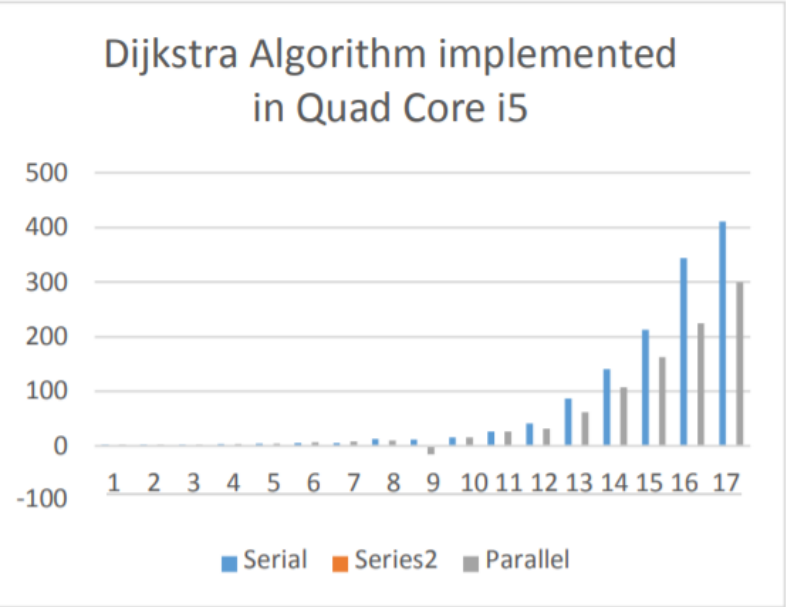
**Output :**


kali@kali: ~/Desktop
File Actions Edit View Help
kali@kali:~/Desktop$ mpicc djik.c
kali@kali:~/Desktop$ mpirun ./a.out
Enter number of vertices in the matrix:
4
0
5
10000
10
10000
0
3
10000
10000
10000
0
1
10000
10000
10000
0
The distance from 0 to each vertex is:
v dist 0→v
1 5
2 8
3 9


kali@kali: ~/Desktop
File Actions Edit View Help
0
1
10000
10000
10000
0
The distance from 0 to each vertex is:
v dist 0→v
1 5
2 8
3 9
The shortest path from 0 to each vertex is:
v Path 0→v
1: 0 1
2: 0 1 2
3: 0 1 2 3
Time elapsed is 0.00134 secondsTime elapsed is 0.00134 secondskali@kali:~/Desk
kali@kali:~/Desktop$ _

## TIME ANALYSIS




| S. No | No. of graph nodes | Execution time (s) | | | | | |
|---|---|---|---|---|---|---|---|
| | | Dijkstra (serial) | Dijkstra (parallel) | Bellman Ford (serial) | Bellman Ford (parallel) | Floyd Warshall (serial) | Floyd Warshall(parallel) |
| 1 | 10 | 0.00112 | 0.002579 | 0.000244 | 0.0068 | 0.0066 | 0.007 |
| 2 | 50 | 0.00381 | 0.004181 | 0.0134 | 0.0183 | 0.003 | 0.018 |
| 3 | 100 | 0.0285 | 0.0268 | 0.0229 | 0.0297 | 0.0175 | 0.05 |
| 4 | 500 | 0.172 | 0.251 | 1.564 | 1.583 | 1.966 | 1.923 |
| 5 | 600 | 0.633 | 0.687 | 2.368 | 3.152 | 2.013 | 2115 |
| 6 | 700 | 0.695 | 0.852 | 3.689 | 4.763 | 3.138 | 3.403 |
| 7 | 800 | 0.81 | 0.919 | 4.905 | 6.545 | 5.26 | 5.409 |
| 8 | 1000 | 0.929 | 0.903 | 10.855 | 10.696 | 23.51 | 23.25 |
| 9 | 2000 | 3.76 | 2.958 | 93.392 | 90.785 | 65.495 | 64.032 |
| 10 | 3000 | 11.431 | 8.833 | 270.372 | 259.179 | 226.893 | 215.826 |
| 11 | 4000 | 23.612 | 17.94 | 768.138 | 754.125 | 511.07 | 451.237 |
| 12 | 5000 | 35.68 | 20.487 | 1985.431 | 1898.604 | 990.99 | 709.583 |
| 13 | 7000 | 76.048 | 52.249 | 5371.892 | 5119.437 | 1845.294 | 9690.312 |
| 14 | 9000 | 124.073 | 109.475 | 13813.532 | 12567.53 | 3901.263 | 2702.129 |
| 15 | 11000 | 186.166 | 165.966 | 29153.981 | 22179.78 | 8130.927 | 6277.424 |
| 16 | 13000 | 362.491 | 229.07 | 71923.419 | 61159.58 | 17259.36 | 9283.12 |
| 17 | 15000 | 482.73 | 310.89 | 145839.83 | 129892.4 | 39129.48 | 26429.42 |

## CONCLUSION

Parallel programming is a very intricate, yet increasingly important task as we have entered the multicore era and more cores are made available to the programmer going from dual core to 32 and 64 core systems or super computers. Independent tasks within a single application can be easily mapped on multicore platforms, the same is not true for applications that do not expose parallelism in a straightforward way. According to the results achieved, Bellman Ford algorithm is a challenging example of such an algorithm that is difficult to accelerate when executed in a multithreaded fashion because of its complexity being O(n^3) . It is very important to have in mind the two major issues inherent to Bellman Ford algorithm: limited independent tasks and excessive synchronization.

There are several possible reasons why the parallel execution of algorithms wasn't as fast as expected for smaller number of nodes. One of the reasons could be that the code used may be inefficient. Another possibility is that, for a small amount of nodes, more time could be spent on parallelization and synchronization then it is spent on execution of code as sequential. Our project report infers that in the efficient transportation system there should be shortest path as well as clean path with no complications in bellman ford and Floyd warshall we see major drawbacks like huge execution time as well as limited independent task . So the Dijkstra set a major foot over them and can be seen as the most efficient solution for the real world applications like transportation system.

**REFERENCES**

[1] Dijkstra E.W. “A note on two problems in connexion with graphs”, in Numerische Mathematik 1, pp.269–271, 1959.

[2] Sniedovich Moshe. “Dijkstra's Algorithm revisited: The Dynamic Programming Connexion”, in Control and Cybernetics vol 35, no. 3, 2006.

[3] Wang Qianyu, Lunhui Xu, and Qiu Jiandong, “Research and realization of the Optimal Path Algorithm with Complex Traffic Regulations in GIS”, Proceedings of the IEEE International Conference on Automation and Logistics, 2007.

[4] R. Dial, F. Glover, D. Karney, and D. Klingman. “A computational analysis of alternative algorithms and labeling techniques for finding shortest path trees", in Network, Vol. 9, No.3, pp. 215-248, 1979.

[5] F. B. Zhan, “Three fastest shortest path algorithms on real road Networks: Data Structures and Procedures”, in Journal of Geographic Information and Decision Analysis, Vol. 1, No.1, pp. 69-82, 1995.

**[6]** Han Cao,Fei Wang, Xin Fang, Hong-lei Tu and Jun Shi, "OpenMP Parallel Optimal Path Algorithm and Its Performance Analysis", World Congress on Software Engineering, 2006

**[7]** Qiang Peng, Yulian Yu and Wenhong Wei, "The Shortest Path Parallel Algorithm on Single Source Weighted Multi-level Graph", 2009 Second International Workshop on Computer Science and Engineering, 2009

**[8]** Nikos Anastopoulos, Konstantinos Nikas, Georgios Goumas and Nectarios Koziris, "Employing Transactional Memory and Helper Threads to Speedup Dijkstra's Algorithm" ,2009

**[9]** Fang Zhou Lin, Nan Zhao, "Parallel Implementation of Dijkstra's Algorithm" , pp

COP5570, 1999

**[10]** Kevin Kelley, Tao B. Schardl, "Parallel Single-Source Shortest Paths", 2007.