PARALLEL IMPLEMENTATIONOFTRAVELLINGSALESMAN PROBLEM

A PROJECT REPORT

Submitted by

Tek Raj Awasthi(17BCE2383)

Riya Shrestha(17BCE2346)

Saurav Khanal(17BCE2345)

CSE4001

Parallel and Distributed Computing

Under the guidance of

Sairabanu J

VIT, Vellore.

SCHOOL OF COMPUTER SCIENCE ANDENGINEERING

NOVEMBER, 2019

INDEX

TITLE:

PARALLEL IMPLEMENTATIONOFTRAVELLINGSALESMAN PROBLEM

ABSTRACT

Travelling Salesman Problem can be solved using greedy algorithmic programs which could be a straightforward, intuitive algorithmic program that's employed in improvement issues. The algorithmic program makes the best selection at every step because it tries to seek out the general best thanks to solve the whole downside. Greedy algorithms area unit quite productive in some issues, like Huffman cryptography that is employed to compress information, or Dijkstra's algorithmic program, that is employed to seek out the shortest path through a graph. Greedy algorithms like Dijkstra's, Bellman Ford and prism algorithms are very much useful to us for solving travelling path problem and solving minimum spanning tree. These algorithms have been developed a long ago. Now days the application of these algorithms are very high. These algorithms are used to find shortest path and saves the time of traveller, specially to the travelling sales man and travel freak people. Because of its huge use the execution time for the algorithms come into the consideration. Also, today we have multicore processors architecture computers so with the intent to reduce the time complexity problem we have used OpenMP API. The theme of the project is to reduce the execution time of these algorithms. So, to reduce the time complexity, these algorithms are parallelly executed inOpenMP.

KEYWORDS:

Travelling Salesman Problem, Dijkstra's Algorithm, Prim's Algorithm, Bellman Ford's Algorithm, Serial Execution, Parallel Execution, OpenMp

Prim's Algorithm:

Prim's algorithm is an algorithm which helps to finds a minimum spanning tree for a weighted undirected graph.it is a greedy algorithm for minimum spanning tree for a

weighted undirected graph. it finds a subset of the edges that forms a tree which include every vertex, where the total weight of all the edges in the tree is minimized. This algorithm performs by constructing tree, one vertex at a time from a starting vertex. At each step adding the least possible connection from the tree to another vertex.

This algorithm can be described as performing the following steps:

- Initialize a tree with a single vertex, chosen arbitrarily from thegraph.
- Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to thetree.
- Repeat step 2 (until all vertices are in thetree).

Dijkstra's Algorithm:

Dijkstra's algorithm is a greedy algorithm which is used for finding the shortest paths between nodes in a graph. This algorithm is used for finding the shortest path between the starting node and every other node. It can also be used for finding the shortest paths from a single node to a single final node by stopping the algorithm once the shortest path to the final node has been determined. For example, each nodes of the graph represents different cities and edge path costs represent total distances between the cities. Thus, Dijkstra's algorithm is used for finding the shortest route between one city and all othercities.

4. Introduction:

4.1. Motivation:

Dynamic Programming is a good approach to try when trying to get an exponential-time algorithm down to polynomial-time.Greedy algorithms are presented for finding a maximal path, for finding a maximal set of disjoint paths in a layered dag, and for finding the largest inducedsubgraphof a graph that has all vertices of degree at leastk.These sequential algorithms can be sped upsignificantlyusingparallelism.This project is aimed to reduce the time complexity of few greedy algorithms (viz. Prism, bellman ford and Dijkstra's algorithm) by the method of dynamic parallel programming. The above algorithms have to be programed dynamically and parallelized with the help of multicore processors. Prism algorithm is MST (minimum cost spanning tree) algorithm and Dijkstra's and bellman ford algorithm are shortest path algorithm.Withthe approach to dynamic programming we are going to propose the dynamic algorithm for the above listed algorithm and ultimately going for parallel implementation of these. Hence, the dynamic algorithm of these greedy algorithms has to bedeveloped.

4.2. Significance:

Parallel and computed programming has immense significance while reducing the time complexity of an execution of the algorithms. With parallelism we can obtain the less execution time for the algorithm. Hence execution time is the major factor of performance and speed. So, to obtain good and suitable speed of processing with open MP we modified the algorithm so that it can be paralleled. Hence it enhances the performance of the algorithm immensely.

4.3. Objective:

ObjectiveWeare dividing the sequential algorithm into parallel algorithm, for this threading concept is used. Program divided into number of threads and each thread is executed independent of other Thread. As number of threads executed

simultaneously, time required to execute that program reduces. Main objective of this project is to save the time required to execute theprograms

5. CONTRIBUTION OFWORK

1. Tek Raj Awasthi – Literature Survey(1,2,3,4,5), Existing system Description, Coding Implementation, Desciption of Modules/Algorithms
2. Saurab Khanal -Literature Survey(6,7,8,9,10), Proposed system Architecture with module wise description, Result and Conclusion
3. Riya Shrestha – Literature Survey(11,12,13,14,15), Introduction, Abstract, Significance, Objective, Overview of work

6. RELATEDWORK

LiteratureSurvey:

5.1 Research Initiatives in Greedy algorithm:

in spite of a serious research by mathematicians, computer scientists, operations researchers, and others, it remains an open question whether or not an efficient general solution exists for the travelling salesman problem. Hence, it is considered as a benchmark problem, and various

methods are being applied to it, so as to get better solutions than those already known. Solutions to the travelling salesman problem can be approached using combinatorial optimization techniques. There are two types of solution search methods: First is to use exact algorithms, which can give optimal solution but takes a huge amount of time. Second is to use approximate algorithms, which never guarantee an optimal solution but gives near optimal solution in a reasonable amount of omputational time. Few of these methods are described below: The most direct solution would be to use the*brute force*approach and try all permutations and check which solution is optimal. The computational complexity of this approach is of the or- der O(N!) for 'N' cities. Because of the ever-increasing number of the possible solutions and the combinatorial nature of the travelling salesman problem, it is impractical to use this approach to solve the problems even with high performance computers.*Insertion heuristics*are also quite straight forward methods, and there are many variants to choose from. The basics of insertion heuristics is to start with a tour of a subset of the cities, and then insert the remaining cities by some heuristic. Tabu Search, created by Fred W. Glover in 1986 and formalized in 1989, is a searching method used for the travelling salesman problem. It is a neighbourhood search

algorithm which searches for better solutions in the neighbourhood of the existing solution. searching methods have a tendency to become stuck in suboptimal regions or on plateaus where many solutions are equally fit. Tabu search increases the performance of these techniques by using memory structures that describe the visited solutions or user-provided sets of rules. This form of memory structures is known as the ‘tabu list’, which consists of a set of rules and banned solutions. The biggest problem with the Tabu search is its running time. Tabu search is only effective in discrete spaces. It is rare that a search would visit the same real-value point in space, multiple times, making the ‘tabu list’ worthless. Also, the ‘tabu list’ can grow very long if the search space is very large or of high dimensionality.

Title	Author	Journal Name	Date of publication	Key concepts	Advantages	Disadvantages	Future Scope
ACO Algorithms for the Traveling Salesman Problem	Thomas STUTZLE and Marco DORIGO	Researchgate	1999	This paper proposed Ant Colony Optimization algorithm. For symmetric TSPs, the distances between the cities are indepe-ndent of the direction of	Each ant has a limited form of memory, called *tabu list,* which allows the ant to retrace its tour, once it is completed. So it results in optimized solution because of updated tour.	Since, these algorithms are in fact hybrid algorithms combining probabilistic solution construction by a colony of ants with standard local search algorithms, such a combination result in a relatively poor solution quality	The application of ACO algorithms to network optimization problems is appealing, since these problems have characteristics like distributed information, non-stationary stochastic dynamics, and

				traversing the arcs. Ants probabilis tica-lly prefer cities which are close and are connecte d by arcs with a high pheromo ne trail strength.		compared to local search algorithms.	asynchronou s evolution of the network status which well match some of the properties of ACO algorithms. like the use of local information to generate solutions, indirect communicati on via the pheromone trails and stochastic state transitions.
SERIAL AND PARALL EL SIMULA TED ANNEA LING AND TABU SEARCH ALGORI THMS FOR THE TRAVEL ING SALESM AN PROBLE M	Miros law MAL EK, Mihi x PAN DYA	Annals of Operati ons Resear ch	1989	This approach uses abbreviat ed cooling schedule and achieves superline speedup. Tabu search has been adopted to execute in parallel computin g environm ent.	This method has a good potential if the TSP is based on real cartographic al map.	During the parallel search, a process may start off with different tour after previous result have been compared. This tour might not correspond to current annealing condition.	While serial implementati on of Tabu search consistently outperforme d serial simulated annealing in terms of cost and time, the performance of parallel incarnations of these algorithms appears to be comparable. So further study should be done to conclude this.

Parameters Analysis Based on Experiments of Particle Swarm Optimization Algorithm for Solving Traveling Salesman Problem	Jiang-wei Zhang	IEEE	2009	LMSK Algorithm: The algorithm works by partitioning the set of tours into smaller sub- tours and compute then parallel.	An improved particle swarm optimization (IPSO) algorithm for solving traveling salesman problem.	The experiment results not only show the effectiveness and efficiency of the proposed method.	The problem with more nodes and can get the global optimum stability and reduce useless searching, all these problems deserve further study.
A combined spatial cluster analysis - traveling salesman problem approach in location-routing problem: A case study in Iran	M S Zaeri, A Marobi	IEEE	2009	Characteristics of location data input for multivariate analysis and calculation of distance matrix have been well utilized.	The method of evaluating all Hamiltonian cycles therefore has exponential order.	Although the PCTSPTW problem is much practical, the related literatures are very limited compared with those on the TSP	The problem with more nodes and can get the global optimum stability and reduce useless searching, all these problems deserve further study.
An Ant Colony Optimization Method for Prizecollecting Traveling Salesman Problem with Time	Xiaohu Shi ; Liupu Wang ; You Zhou ; Yanchun Liang	IEEE	2008	This approach uses abbreviated cooling schedule and achieves superline speedup. Tabu search	This method has a good potential if the TSP is based on real cartographical map.	Although the PCTSPTW problem is much practical, the related literatures are very limited compared with	. This method has a good potential if the TSP is based on real cartographical map.

Windows				has been adopted to execute in parallel computing environment.		those on the TSP.	
Parallelizing GA Based Heuristic approach for TSP over CUDA and OpenMP	Rahul Saxena, Monika Jain,	IEEE	2007	A number of heuristic approaches have been proposed for the system. These are referred to as tour construction approaches which halts when an optimal tour is found. The simplest and the most straightforward heuristic proposed was closest neighbor heuristic.	CUDA implementation takes the advantage of the fact that due to integration of many integrated cores so called coprocessors. This helps in cutting down the execution time drastically as throughput ratio has been improved or in other words, *Compute to Global Memory Access* (CGMA) has been improved significantly.	But the computational complexity of the problem was high in the case approximately of order of O (n2 2n-1) and hence efficient results could be produced only till 11 cities.	By implementing the solution in parallel over CUDA, the execution time to find the optimal or near-optimal solution decreases by 7 times. The idea can be further extrapolated to a network environment where fast topology building is a necessity to cope up with the real time scenarios in Mobile Ad-hoc Networks (MANET) and Vehicular Adhoc Networks (VANET)

							over a huge complex network of devices for efficient routing.
An Improved Routing Optimization Algorithm Based on Travelling Salesman Problem for Social Networks	Naixue Xiong, Chunxu-e Wu	Sustain a-bility	2007	The more ants on the path, the more pheromones are secreted, and the path will be chosen by more ants. On the contrary, the fewer ants on the path, the less pheromone are secreted, the fewer ants will choose the path, so most of the ants will choose the path of pheromone concentration to find food.	Compared with other algorithms, the ant colony algorithm has the characteristics such as good distributed computing mechanism and strong robustness, so it can be combined with other algorithms to put forward a more powerful algorithm.	The pherom-one left by the ant colony in the first cycle is not necessarily the optimal direction of the path. The effect of positive feedback leads to the enhancement of the information on the non-optimal path and the hindrance of the global optimal solution.	The application depth of ant colony algorithm is not enough because most simulation experiments are carried out under specific experimental conditions, while the actual situation is dynamic. Thus, the relevant issues have yet to be further expanded.
Research on Solving	Muhao Chen,	IEEE XPLORE	2015	The simplest and the	Annealing algorithm can avoid the	GA has the problem of prematurity,	It shows that by using the genetic

Traveling Salesman Problem Based on Virtual Instrument Technology and Genetic-Annealing Algorithms	Chen Gong,			most straightforward heuristic proposed was closest neighbor heuristic.	weakness of genetic algorithm due to its extraordinary ability of local optimization.	local optimization will appears to be peaky and poor because the disadvantages of randomness of this algorithm. Therefore, though the result is near to the best solution, the error caused by randomness cannot be effectively avoided.	algorithm to find out the global optimization and then the annealing algorithm to find out the local optimization can obtain the best optimized result. The result proves the practicability and accuracy of this method.

Overview of the Work:

1. Problemdescription:

The travelling salesman problem (TSP) is a popular mathematics problem that seeks for the most feasible path possible given a set of the points and cost that must all be visited. In computer science, the problem seeks the most efficient route to travel between various nodes, for data. In terms of input, the problem takes a list of physical locations or system nodes, along with distance information. Algorithms and equations work on the process of identifying the most efficient paths possible between the locations. Computer programs can do this through the process of elimination or through a process called heuristics that provides probability outcomes for this type ofequation.

2.Drawbacks:

The simplest attitude to solve the problem of travelling salesman is to try all the 12 possible routes/all possible combinations of cities to visit while summing distances between the cities in particular route and finally find the route with shortest total distance. This very straight-forward solution is called brute-force or exhaustive search. However, this technique has a shortcoming of immense importance running time of $\Theta((n-1)!)$, i.e. it needs to generate (n-1)! Permutations of n cities and calculate the total distance of it. And as n grows, the factorial (n-1)! becomes larger than all polynomials and exponential functions (but slower than double exponential functions) in n-1. This enormous growing of possible routes means enormous growing of time needed for solving TSP even for contemporarycomputers.

Due to its time complexity $\Theta(N!)$ this algorithm is not very suitable for the purpose of this work and so if there exists other algorithm with better time complexity but still

exact (thus returning the route of the same quality as brute-force algorithm, i.e. optimal route), it should
be chosen for the implementation rather than brute force.

- Cuttingplanes:

The basic idea behind cutting planes is to set constraints to a linear program until the best basic possible efficient solution takes on integer values. There are two ways to generate cuts. The first,

called Gnomery cuts, generates cuts from any linear programming tableau. This has the disadvantage that the method can be very slow. The next approach is to use the structure of the problem to generate appropriate cuts. The approach needs a problem-by-problem analysis.

•Heuristic algorithms-Greedyalgorithm:

In this heuristic, an instance as a complete graph with the cities as vertices and with an edge of length d (ci, cj) between each pair {ci, cj} of cities can be viewed. A tour is then simply a Hamiltonian cycle in this graph, i.e., a connected group of edges in which every city has degree 2. This cycle is built up with one edge at a time, starting with the shortest edge, and repeatedly adding the shortest remaining available edge, where an edge is accessible if it is not yet in the tour and if adding it would not create a cycle of length less than N or a degree-3 vertex. The Greedy heuristic can be implemented to run in time Θ (N 2 log N) and is thus somewhat slower than Nearest Neighbour.

• Heuristic algorithms-Simulatedannealing:

The annealing process begins with a material in a melted state and then gradually lowers its temperature, analogous to decreasing an objective function value by a series of improving moves. However, in the physical setting the temperature must not be lowered toorapidly,particularly in its early stages. Otherwise certain locally suboptimal configurations can be frozen into the material and the ideal/optimal low energy state will not be reached.Toallow a temperature to move slowly through a certain region maps to permitting non-improving moves to be selected with a certain probability – a probability which reduces as the energy level (objective function value) of the system diminishes. Thus, in the analogy to combinatorial problem solving as stated in, it is postulated that the path to an optimal state likewise begins from one of diffuse randomization, somewhat removed from optimality, where non-improving moves are initially accepted with a relatively high probability which is gradually decreased over time. Unfortunately, the time complexity of simulated annealing algorithm cannot be easily found, there exist only experimental estimations which are still not usable for all TSP scenarios and cannot be generally used. For this

reason, simulated annealing doesn't fit very well to our situation and required goal where I need to calculate the solution within limited time.

3. SoftwareRequirements:

- Gcccompiler
- Open MP

4. HardwareRequirements:

- Intel core 2duo

9. Proposed system architecture with module wise

SYSTEMARCHITECTURE(DESIGN)

The main goal of a parallel program is to utilize the multicore resource variable in common for improving the performance of algorithm. The paper solutions of which takes more time using sequential algorithm on a single processor machine or on multiprocessor machine. The fast solution of these problems can be obtained using parallel algorithms and multicore system. In greedy algorithm, there is no task dependency hence thread and kernel instances parallel running reduces execution time using openMP.

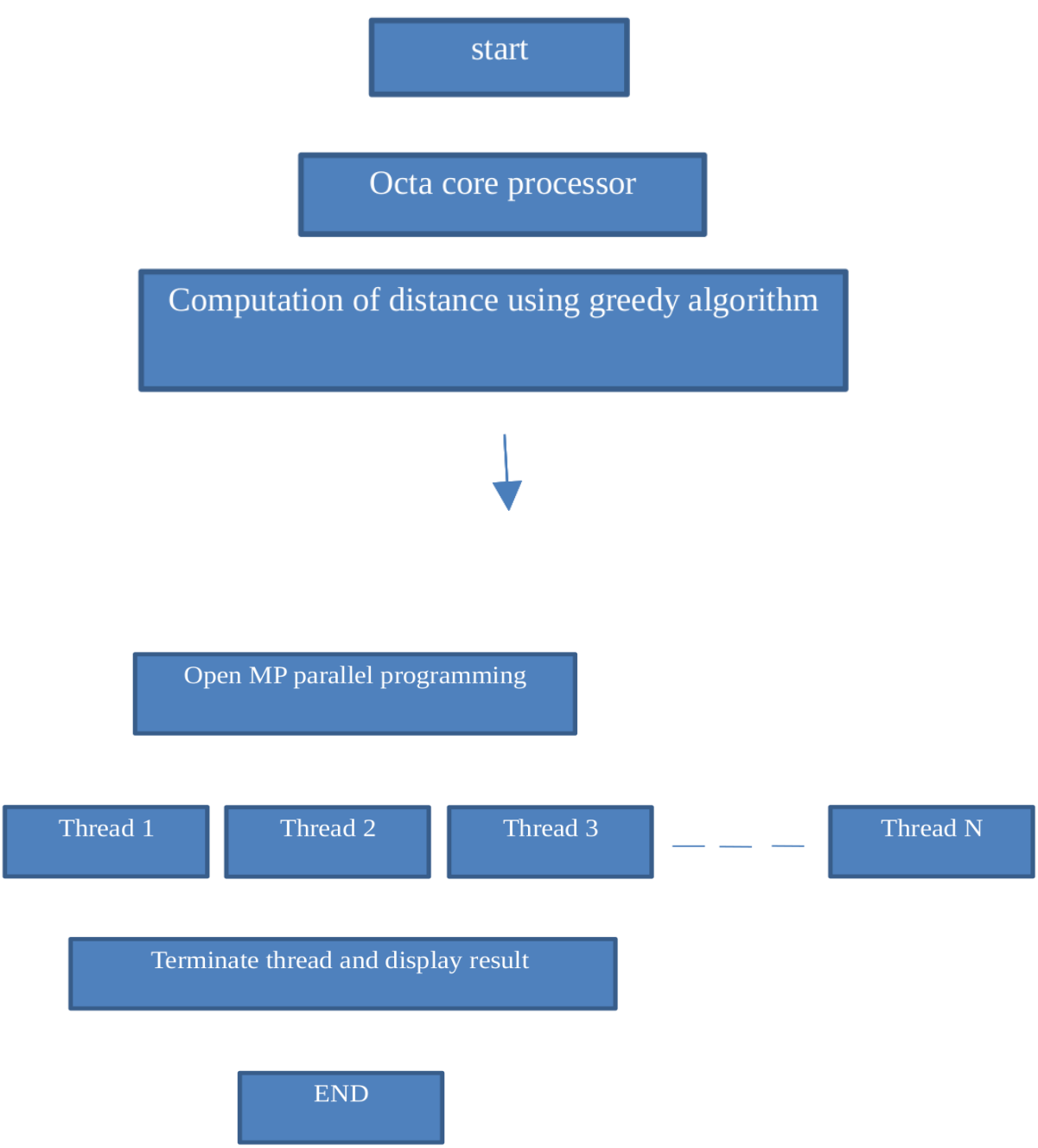

start
Octa core processor
Computation of distance using greedy algorithm
Open MP parallel programming
Thread 1
Thread 2
Thread 3
Thread N
Terminate thread and display result
END

Fig. 1. Flow chart of OpenMP Module

Description of Modules/Programs

9.1 Prim'sAlgorithm:

Prim's algorithm is an algorithm which helps to finds a minimum spanning tree for a weighted undirected graph.it is a greedy algorithm for minimum spanning tree for a weighted undirected graph. it finds a subset of the edges that forms a tree which include every vertex, where the total weight of all the edges in the tree is minimized. This algorithm performs by constructing tree, one vertex at a time from a starting vertex. At each step adding the least possible connection from the tree to another vertex.

This algorithm can be described as performing the following steps:

- Initialize a tree with a single vertex, chosen arbitrarily from thegraph.
- Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to thetree.
- Repeat step 2 (until all vertices are in the tree).

The time complexity of Prim's algorithm is O ((V + E) l o g V) and depends on the data structures used for the graph and edges weight, which can be perform using a priority queue.

9.2 Dijkstra'sAlgorithm:

Dijkstra's algorithm is a greedy algorithm which is used for finding the shortest paths between nodes in a graph. This algorithm is used for finding the shortest path between the starting node and every other node. It can also be used for finding the shortest paths from a single node to a single final node by stopping the algorithm once the shortest path to the final node has been determined. For example, each nodes of the graph represents different cities and edge path costs represent total distances between the cities. Thus, Dijkstra's algorithm is used for finding the shortest route between one city and all othercities.

- Dijkstra's algorithm assigns some initial distance values and tries to improve them step by step-
- Mark all nodes unvisited. Creating a set of all unvisitednodes.
- Assign every node with initial distance value- set it to zero for our initial node and to infinity for all other nodes. Set the initial node ascurrent.
- considering the current node, mark all of its unvisited neighbours and find the tentative distances of all unvisited node through the current node and Compare it with the newly calculated tentative distance to the current calculated value and compute the leastone.
- considering all unvisited neighbours' node of the current node and marking the current node asvisitedandremoveit fromthe unvisited set.Avisitednode willneverbechecked again.
- If the final node has been marked as visited or if the smallest tentative values among the nodes in the unvisited set isinfinity,then stop. The algorithm hasfinished.
- Otherwise, selecting the unvisited node which is marked as the smallest tentative distance, set it as the new "current node", and go back to step3.

The shortest path algorithm called Dijkstra's algorithm is implemented and presented in parallel way so that to reduce time complexity. The algorithm implementation was parallelized using OpenMP (Open Multi-Processing) standards. Its performances were measured on different configurations, based on quad core and i5 processors. The experimental results show that the parallel execution of the algorithm has good performances in terms of speed-up ratio compared to its serial execution. Dijkstra's

algorithm itself is sequential, and difficult to parallelize so average speed-up ratio achieved by parallelization is only 10% which is a huge disadvantage of this algorithm, because its use is widespread, and enhancing its performance would have great effects in its many uses. This algorithm stops once the final node has the smallest tentative distance among all "unvisited"nodes.

9.3 BellmanFord:

The Bellman-Ford algorithm is a process to computes shortest paths, to all of the other vertices in a weighted digraph, from a single source vertex. It is slower compared to Dijkstra's algorithm for the same problem. However, it more versatile, as it is capable of handling graphs for negative edges too. Negative edge weights can be found in several applications of graphs, hence the usefulness of this algorithm. If a graph having a "negative cycle" i.e. a cycle whose cost sum of edges is a negative value, that is reachable from the source, then cheapest path cannot be calculated because any path having a node on the negative cycle can be made cheaper by one more retrace around the negative cycle. In such a case, the Bellman Ford algorithm detects negative cycles and mark their existence.

Like Dijkstra's Algorithm, Bellman–Ford also follow the approach of relaxation, in which an approximation to the optimal path is gradually corrected by more accurate values until eventually tending the optimum solution. In both algorithms, the approximate cost to each vertex is always an overestimate of the actual cost, and is replaced by the lesser known old value and the cost of a newly found path. However, Dijkstra's algorithm uses a priority queue to select the closest vertex greedily that has yet to be processed, and execute this relaxation process on all of its outgoing edges; by contrast, the Bellman-Ford algorithm simply relaxes all the edges, and repeat |V|-1 times, where |V| is the number of vertices in the graph. In each repetition, the number of vertices with accurately calculated costs grows, from which it follows that finally all vertices will have their accurate cost. This approach allows the Bellman-Ford algorithm to be applied to a wide range of class of inputs than Dijkstra. Bellman-Ford runs in O (| V|. |E|) time, where |V| and |E| represents the number of vertices and edgesrespectively.

Steps:

- The algorithm initializes the cost to the source to 0 and all remaining nodes toinfinity.
- For all edges, if the cost to the final node can be shortenedby
- taking the edge, the cost is updated to the new lowervalue.
- At each and every iteration i the edges are retraced, the algorithmsorts

- all smallest paths of at most length i edges. Since the longest possible path without a cycle can be |V|-1 edges, the edges must be retraced |V|-1 times to ensure the shortest path for all nodes.
- A final retrace of all the edges is performed and if any distance is updated, then a path of length |V| edges has been found which can only occur if at least one negative cycle appears in thegraph.

10. Source Code andresult

a) forprism’salgorithm:

```
#include<stdio.h>
#include<omp.h>
#include<time.h>
#include<unistd.h>

int main()
{
clock_t begin = clock();

int n;
printf("Enter No Of Vertices : ");
scanf( "%d" , &n );
int a[n][n];
int nv[n];
int v[n];
int i , j , k;
int over = 0;
int min = 1000000;
int vertex;
int mstCost = 0;
for( i = 0 ; i < n ; i++)
{
for( j = 0 ; j < n ; j++)
{
printf( "Enter Edge Weight ( %d , %d ) : " , i , j );
scanf( "%d" , &a[i][j] );
}
nv[i] = i;
```

```
v[i] = -1;
printf( "nv[i]=%d \n",nv[i]);
}
if( n > 0 )
{
v[0] = nv[0];
nv[0] = -1;
}
//-1 indicates node is visited
while( over == 0 )
{
omp_set_num_threads(4);
#pragma omp parallel for
for( i = 0 ; i < n ; i++ )
{
if( v[i] != -1 )
{
for( j = 0 ; j < n ; j++ ){
if( nv[j] != -1 )
{
if( min > a[v[i]][nv[j]] && a[v[i]][nv[j]] != 0 )
{
min = a[v[i]][nv[j]];
vertex = nv[j];
k = j; //for indicates visit
}
}
}
}
}
```

```
printf( "min=%d \n",min);
mstCost += min;
min = 1000000;
nv[k] = -1; //visited
for( i = 0 ; i < n ; i++ )
{
if( v[i] == -1 )
{
v[i] = vertex;
break;
}
}
over = 1;
for( i = 0 ; i < n ; i++ )
{
if( nv[i] != -1 )
{
over = 0;
break;
}
}
}
printf( "MST COST : %d \n" , mstCost);

clock_t end = clock();
double time_spent = (double)(end-begin)/CLOCKS_PER_SEC;
printf("Time spent: %lf\n", time_spent);
```

```
return 0;
}
```

b) for Dijkstra'salgorithm:

```
#include <stdio.h> #include
<omp.h> #include<algorithm>

#define INFINITY 100000
int V,E;

//Structure for vertex
typedef struct
{
	int label;
	bool visited;

} Vertex;

//Structure for directed edge from u to v
typedef struct
{
	int u;
	int v;

} Edge;

//Printing Shortest Path Length
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");
        }
}



//Finds weight of the edge that connects Vertex u with Vertex v
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];
                }
        }
        // If no edge exists, weight is infinity
        return INFINITY;
}
```

```
//Get the minimum path length among the paths
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(vertices[i].visited == false && 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 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];

        // Mark first vertex as visited, shortest path =
        0 root->visited = true;
        path_length[root->label] = 0;
        int i, j;
        // Compute distance to other vertices
        for(i = 0; i < V;i++)
        {

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

```
        else
        {

            vertices[i].visited = true;
        }
    }

    parallel_start = omp_get_wtime();
    // External For Loop
    for(j = 0; j < V; j++)
    {
        Vertex u;
        // Obtain the vertex which has shortest distance and mark it as visited
        int h = minimimPathVertex(vertices, path_length);
        u = vertices[h];

        //Update shortest path wrt new source
        //Internal For Loop, Parallelising the computation
        #pragma omp parallel for schedule(static) private(i)
        for(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);
            }
        }
    }
    parallel_end = omp_get_wtime();
```

```
    printShortestPathLength(path_length);
    printf("\nRunning time: %lf ms\n", (parallel_end - parallel_start)*1000);
}

int main()
{
    printf("Enter number of vertices: ");
    scanf("%d",&V);
    printf("Enter number of edges: ");
    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;
    }

    printf("\nEnter these details \nFROM \tTO \tWEIGHT\n");
    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;
	printf("\nEnter Source Vertex: ");
	scanf("%d",&source);
	Vertex root = {source, false};

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

	return 0;
}
```

c)for bellman ford:

```
//BELLMAN_FORD

#include <string>
#include <cassert>
#include <iostream>
#include <fstream>
#include <algorithm>
#include <iomanip>
#include <cstring>

#include "mpi.h"

using std::string;
using std::cout;
using std::endl;

#define INF 1000000
```

```
/**
 * utils is a namespace for utilityfunctions
 * including I/O (read input file and print results) and matrix dimensionconvert(2D->1D) function
 */
namespace utils {
    int N; //number of vertices
    int *mat; // the adjacency matrix

    void abort_with_error_message(string msg) {
        std::cerr << msg << endl;
        abort();
    }

    //translate 2-dimension coordinate to 1-dimension
    int convert_dimension_2D_1D(int x, int y, int n) {
        return x * n + y;
    }

    int read_file(string filename) {
        std::ifstream inputf(filename, std::ifstream::in);
        if (!inputf.good()) {
            abort_with_error_message("ERROR OCCURRED WHILE READING INPUT FILE");
        }
        inputf >> N;
        //input matrix should be smaller than 20MB * 20MB (400MB, we don't have too much memory for multi-processors)
        assert(N < (1024 * 1024 * 20));
        mat = (int *) malloc(N * N * sizeof(int));
```

```
        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++) {
                inputf >> mat[convert_dimension_2D_1D(i, j, N)];
            }
        return 0;
    }

    int print_result(bool has_negative_cycle, int *dist) {
        std::ofstream outputf("output.txt", std::ofstream::out);
        if (!has_negative_cycle) {
            for (int i = 0; i < N; i++) {
                if (dist[i] > INF)
                    dist[i] = INF;
                outputf << dist[i] << '\n';
            }
            outputf.flush();
        } else {
            outputf << "FOUND NEGATIVE CYCLE!" << endl;
        }
        outputf.close();
        return 0;
    }
}//namespace utils


/**
 * Bellman-Ford algorithm. Find the shortest path from vertex 0 to othervertices.
 * @param my_rank the rank of currentprocess
 * @param p number ofprocesses
 * @param comm the MPIcommunicator
```

```
 * @param n inputsize
 * @param *mat input adjacency matrix
 * @param *dist distancearray
 * @param *has_negative_cycle a bool variable to recode if there are negativecycles
*/
void bellman_ford(int my_rank, int p, MPI_Comm comm, int n, int *mat, int *dist,
bool *has_negative_cycle) {
    int loc_n; // need a local copy for N
    int loc_start, loc_end;
    int *loc_mat; //local matrix
    int *loc_dist; //local distance

    //step 1: broadcast N
    if (my_rank == 0) {
        loc_n = n;
    }
    MPI_Bcast(&loc_n, 1, MPI_INT, 0, comm);

    //step 2: find local task range
    int ave = loc_n / p;
    loc_start = ave * my_rank;
    loc_end = ave * (my_rank + 1);
    if (my_rank == p - 1) {
        loc_end = loc_n;
    }

    //step 3: allocate local memory
    loc_mat = (int *) malloc(loc_n * loc_n * sizeof(int));
    loc_dist = (int *) malloc(loc_n * sizeof(int));
```

```
    //step 4: broadcast matrix mat
    if (my_rank == 0)
        memcpy(loc_mat, mat, sizeof(int) * loc_n * loc_n);
    MPI_Bcast(loc_mat, loc_n * loc_n, MPI_INT, 0, comm);

    //step 5: bellman-ford algorithm
    for (int i = 0; i < loc_n; i++) {
        loc_dist[i] = INF;
    }
    loc_dist[0] = 0;
    MPI_Barrier(comm);

    bool loc_has_change;
    int loc_iter_num =0;
    for (int iter = 0; iter < loc_n - 1; iter++) {
        loc_has_change = false;
        loc_iter_num++;
        for (int u = loc_start; u < loc_end; u++) {
            for (int v = 0; v < loc_n; v++) {
                int weight = loc_mat[utils::convert_dimension_2D_1D(u, v, loc_n)];
                if (weight < INF) {
                    if (loc_dist[u] + weight < loc_dist[v]) {
                        loc_dist[v] = loc_dist[u] + weight;
                        loc_has_change = true;
                    }
                }
            }
        }
        MPI_Allreduce(MPI_IN_PLACE, &loc_has_change, 1, MPI_CXX_BOOL,
MPI_LOR, comm);
```

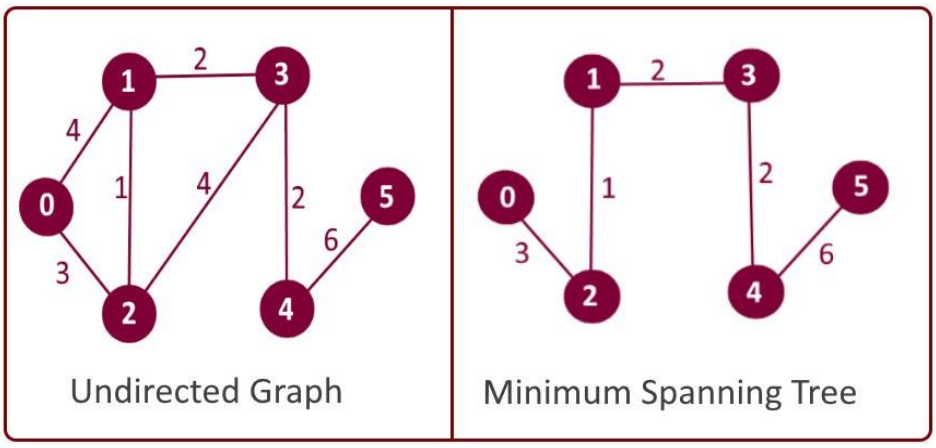
```
        if (!loc_has_change)
            break;
        MPI_Allreduce(MPI_IN_PLACE, loc_dist, loc_n, MPI_INT, MPI_MIN,
comm);
    }

    //do one more step
    if (loc_iter_num == loc_n - 1) {
        loc_has_change = false;
        for (int u = loc_start; u < loc_end; u++) {
            for (int v = 0; v < loc_n; v++) {
                int weight = loc_mat[utils::convert_dimension_2D_1D(u, v, loc_n)];
                if (weight < INF) {
                    if (loc_dist[u] + weight < loc_dist[v]) {
                        loc_dist[v] = loc_dist[u] + weight;
                        loc_has_change = true;
                        break;
                    }
                }
            }
        }
        MPI_Allreduce(&loc_has_change, has_negative_cycle, 1, MPI_CXX_BOOL,
MPI_LOR,comm);
    }

    //step 6: retrieve results back
    if(my_rank ==0)
        memcpy(dist, loc_dist, loc_n * sizeof(int));

    //step 7: remember to free memory
```

```
    free(loc_mat);
    free(loc_dist);

}

int main(int argc, char **argv) {
    if (argc <= 1) {
        utils::abort_with_error_message("INPUT FILE WAS NOT FOUND!");
    }
    string filename = argv[1];

    int *dist;
    bool has_negative_cycle = false;

    //MPI initialization
    MPI_Init(&argc, &argv);
    MPI_Comm comm;

    int p;//number of processors
    int my_rank;//my global rank
    comm = MPI_COMM_WORLD;
    MPI_Comm_size(comm, &p);
    MPI_Comm_rank(comm, &my_rank);

    //only rank 0 process do the I/O
    if (my_rank == 0) {
        assert(utils::read_file(filename) == 0);
        dist = (int *) malloc(sizeof(int) *utils::N);
    }
```

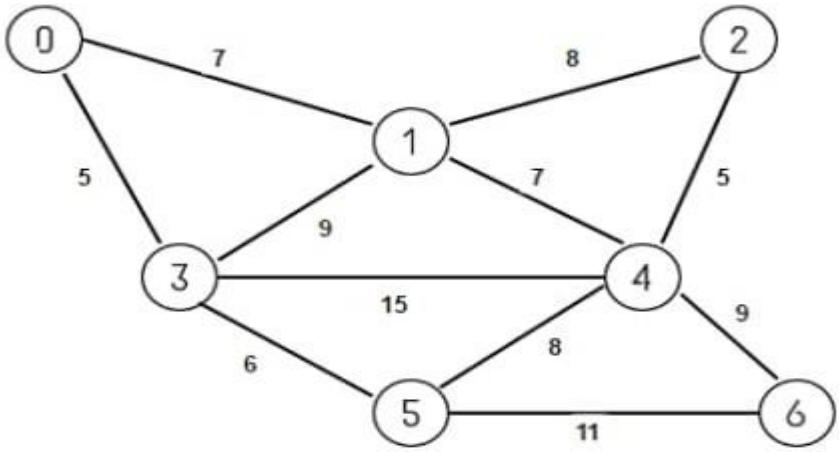
```
    //time counter
    double t1, t2;
    MPI_Barrier(comm);
    t1 = MPI_Wtime();

    //bellman-ford algorithm
    bellman_ford(my_rank, p, comm, utils::N, utils::mat, dist, &has_negative_cycle);
    MPI_Barrier(comm);

    //end timer
    t2 = MPI_Wtime();

    if (my_rank == 0) {
        std::cerr.setf(std::ios::fixed);
        std::cerr << std::setprecision(6) << "Time(s): " << (t2 - t1) << endl;
        utils::print_result(has_negative_cycle, dist);
        free(dist);
        free(utils::mat);
    }
    MPI_Finalize();
    return 0;
}
```

5.3. Test cases

a) forprism’salgorithm:



b) for Dijkstra'salgorithm:



c)for bellman ford:

input1.txt

10. RESULTS

10.1 Execution snapshots

a) forprism'salgorithm:

```
bibek@BIBEK:~$ gedit prism.c
bibek@BIBEK:~$ gcc -o prism -fopenmp prism.c
bibek@BIBEK:~$ ./prism
Enter No Of Vertices : 6
Enter Edge Weight ( 0 , 0 ) : 0
Enter Edge Weight ( 0 , 1 ) : 4
Enter Edge Weight ( 0 , 2 ) : 3
Enter Edge Weight ( 0 , 3 ) : 0
Enter Edge Weight ( 0 , 4 ) : 0
Enter Edge Weight ( 0 , 5 ) : 0
nv[i]=0 Enter Edge Weight ( 1 , 0 ) : 4
Enter Edge Weight ( 1 , 1 ) : 0
Enter Edge Weight ( 1 , 2 ) : 1
Enter Edge Weight ( 1 , 3 ) : 2
Enter Edge Weight ( 1 , 4 ) : 0
Enter Edge Weight ( 1 , 5 ) : 0
nv[i]=1 Enter Edge Weight ( 2 , 0 ) : 3
Enter Edge Weight ( 2 , 1 ) : 1
Enter Edge Weight ( 2 , 2 ) : 0
Enter Edge Weight ( 2 , 3 ) : 4
Enter Edge Weight ( 2 , 4 ) : 0
Enter Edge Weight ( 2 , 5 ) : 0
nv[i]=2 Enter Edge Weight ( 3 , 0 ) : 0
Enter Edge Weight ( 3 , 1 ) : 2
Enter Edge Weight ( 3 , 2 ) : 4
Enter Edge Weight ( 3 , 3 ) : 0
Enter Edge Weight ( 3 , 4 ) : 2
Enter Edge Weight ( 3 , 5 ) : 0
nv[i]=3 Enter Edge Weight ( 4 , 0 ) : 0
Enter Edge Weight ( 4 , 1 ) : 0
Enter Edge Weight ( 4 , 2 ) : 0
Enter Edge Weight ( 4 , 3 ) : 2
Enter Edge Weight ( 4 , 4 ) : 0
Enter Edge Weight ( 4 , 5 ) : 6
nv[i]=4 Enter Edge Weight ( 5 , 0 ) : 0
Enter Edge Weight ( 5 , 1 ) : 0
Enter Edge Weight ( 5 , 2 ) : 0
Enter Edge Weight ( 5 , 3 ) : 0
Enter Edge Weight ( 5 , 4 ) : 6
Enter Edge Weight ( 5 , 5 ) : 0
nv[i]=5 min=3
min=1
min=2
min=2
min=6
MST COST : 14
bibek@BIBEK:~$
```

b) for Dijkstra'salgorithm:

```
bibek@BIBEK:~/Desktop/project$ g++ -std=c++11 -fopenmp dkk.cpp -o dkk
bibek@BIBEK:~/Desktop/project$ ./dkk
Enter number of vertices: 7
Enter number of edges: 11

Enter these details
FROM    TO      WEIGHT
0       1       7
0       3       5
1       2       8
1       3       9
1       4       7
2       4       5
3       4       15
3       5       6
4       6       9
4       5       8
5       6       11

Enter Source Vertex: 0

VERTEX  SHORTEST PATH LENGTH
0       0
1       7
2       15
3       5
4       14
5       11
6       22

Running time: 3.305597 ms
bibek@BIBEK:~/Desktop/project$
```

c) for Bellman Ford

input1.txt

```
bibek@BIBEK:~/Desktop/project$ gedit mpi_bellman_ford.cpp
bibek@BIBEK:~/Desktop/project$ mpic++ -std=c++11 -o mpi_bellman_ford mpi_bellman_ford.cpp
bibek@BIBEK:~/Desktop/project$ mpiexec -n 2 ./mpi_bellman_ford input2.txt
Time(s): 0.086407
bibek@BIBEK:~/Desktop/project$ mpiexec -n 4 ./mpi_bellman_ford input2.txt
Time(s): 0.075454
```

11. Conclusion and Future Directions:

After studying all the techniques to solve greedy algorithm, it is concluded that the traditional algorithms have major shortcomings:firstly,they are not suitable for negative edge networks;secondly,they exhibit higher computational complexity. Therefore, using developed ACO algorithm, a group of ants can effectively explore the graph and generate the optimal solution. Although, researchers have got remarkable success in designing a better algorithm in term of space and time complexity to solve shortest path problem. But ACO for greedy algorithm has efficient and can solve the problem in short time. The performance of ACO algorithm depends on the appropriate setting of parameters. These parameters depend on the problem instance in hand and also on the required solution accuracy

12. ACKNOWLEDGEMENT

We would like to express our special thanks and gratitude to our professor and the

project guideProf.Sairabanu Jas well as VIT who gave us this golden opportunity to work on this wonderful project on the topic "**PARALLEL IMPLEMENTATION OF TRAVELLING SALESMAN PROBLEM"**which also helped us in doing a lot of research and we came to know about so many new things, we are
really thankful to them.
We would also like to thank all the participants of our survey to co-operate with us and gave their precious time.

Place: Vellore

13. References:

[1] Cao H,WangF,Fang X,TuH-L, Shi J. OpenMP parallel optimal path algorithm and its performance analysis. 2009 WRIWorldCongr Softw Eng WCSE 2009.2009;1.

[2] AwariR. Parallelization of shortest path algorithm using OpenMP and MPI. Proc Int ConfIoT Soc Mobile, Anal Cloud, I-SMAC 2017.2017;304–9.

[3] Pathare S, KulkarniP,Kardel R. Performance Analysis of Algorithm UsingOpenMP.2014;1(1):152–6.

[4] Fallis A. Data Structure and algorithms in java.Vol.53, Journal of Chemical Informationand Modeling. 2013. 1689-1699 p.

[5] Prim RC. Shortest Connection Networks And Some Generalizations. Bell SystTechJ.1957;36(6):1389-401.

[6] Grama A, Gupta A, Karypis G, KumarV.Introduction to Parallel Computing, SecondEdition.Communication.2003.856p.

[7] P.Pacheco, An Introduction to Parallel Programming, Burlington, USA:Elsevier,2011.

[8] ShiY,Lu J, Shi X, Zheng J, Li J. The Key Algorithms of Promoter Data ParallelProcessingbasedonOpenMP.2014;154–7.

[9] OpenMP ArchitectureReviewBoard,"OpenMP ApplicationProgramInterface ",http://openmp.org/forum/

[10] AnjaneyuluGSGN,DashoraR,VijayabarathiA,RathoreBS.Improvingtheperformanceof Approximation algorithm to solve Travelling Salesman Problem using Parallel Algorithm. 2014;337(3):334–7.

[11] Jasika N, Alispahic N, Elma A, Ilvana K, Elma L, Nosovic N. Dijkstra's shortest path algorithm serial and parallel execution performance analysis. MIPRO, 2012 Proc 35th Int Conv Inf CommunTechnolElectron Microelectron. 2012;1811–5.

[12] Maroosi A, Muniyandi RC, Sundararajan E, Zin AM. Parallel and distributed computing models on a graphics processing unit to accelerate simulation of membrane systems. Simulation Modelling Practice andTheory.2014;47:60-78.Availablefrom, DOI:10.1016/j.simpat.2014.05.005

[13] Ang MC, Aghamohammadi A, NgKW,Sundararajan E, Mogharrebi M, LimTL. Multi-coreFrameworksInvestigationonareal-timeobjectTrackingapplication.JTheorAppl InfTechnol.2014;70(1):163