PARALLEL ALGORITHMS FOR SHORTEST PATH PROBLEM ON TIME DEPENDENT GRAPHS

by

Mehmet Akif Ersoy

B.S., Computer Engineering, Boğaziçi University, 2012

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering
Boğaziçi University
2015

PARALLEL ALGORITHMS FOR SHORTEST PATH PROBLEM
ON TIME DEPENDENT GRAPHS

APPROVED BY:

Prof. Can Özturan (Thesis Supervisor)	....
Assist. Prof. Ali Haydar Özer	....
Assoc. Prof. Haluk O. Bingöl	....

DATE OF APPROVAL: 16.12.2015

ACKNOWLEDGEMENTS

I would like to thank my thesis supervisor Can Özturan for his guidance and passion during this research. His vision and expertise in this field made this work possible.

I would also like to thank Assoc. Prof. Dr. Haluk O. Bingöl and Asst. Prof. Dr. Ali Haydar Özer for their participation in my thesis jury and their precious feedbacks.

I am also thankful to my colleagues at TÜBİTAK BİLGEM for their encouragements and making the working environment such a great place to work. I have learned a lot from them about work and non-work related subjects.

Last but not least, I would like to thank my family. First, I want to thank my twin brother, Oğuzhan, for always being there for me. I could not think my life without him. I would like to thank my sister, Esra, for always keeping an eye on me and never withholding her assistance when I need her. I want to thank my father, İbrahim, whom I owe everything that I have now and will have in the future. I have learned many things from him that are too many to count here. Lastly, I would like to dedicate this thesis to my mother, Ayşe, who raised me with unconditional love and care, and I will always feel her best wishes in my entire life.

ABSTRACT

PARALLEL ALGORITHMS FOR SHORTEST PATH PROBLEM ON TIME DEPENDENT GRAPHS

Shortest path problem in time dependent graphs has become a popular problem in recent years. Ever since the smart phones became an inseparable part of our lives, the applications on those devices started to provide many functionalities which make human life much easier. Navigation applications are one of them. State of the art navigation applications benefit from real time traffic data besides the map data. Therefore, it becomes a necessity to solve the problem of shortest path with real time data, i.e., on time dependent graphs.

Various sequential algorithms for the shortest path problem in time dependent graphs are appearing in the literature. However, these algorithms mostly suffer from the following two problems: long running times or huge memory requirements. These problems of the previously proposed algorithms are making them unsuitable for navigation applications which run on real time data and which need fast response times. In order to speed-up the running time of the sequential algorithm, without requiring much more memory, for shortest path problem with time dependent flow speed model, we propose parallel algorithms based on Modified Dijkstra algorithm. We develop three different parallel implementations by using Cuda and OpenMP: These are (i) a Cuda based version, (ii) an OpenMP based version and (iii) a hybrid Cuda and OpenMP based version. We get up to 10-fold speedup in the OpenMP version, and 17-fold speed up in the other two versions.

ÖZET

ZAMANA BAĞIMLI ÇİZGELERDE EN KISA YOL PROBLEMİ İÇİN PARALEL ALGORİTMALAR

Zamana bağımlı çizgeler için en kısa yol problemi son yıllarda oldukça popülerleşti. Akıllı telefonlar hayatımızın ayrılamaz bir parçası olduğundan beri, bu telefonlardaki uygulamalar insan hayatını kolaylaştıran birçok olanak sağlıyor. Navigasyon uygulamaları da bunlardan biridir. Son teknoloji ürünü olan yer bulma uygulamaları harita verisinin yanında gerçek zamanlı trafik verilerinden de yararlanmaktadırlar. Dolayısıyla, en kısa yol problemini gerçek zamanlı verilerde, yani zamanla değişen çizgelerde çözmek artık bir gereklilik olmuştur.

Zamana bağımlı çizgelerde en kısa yol problemi için literatürde çeşitli ardışıl algoritmalar bulunmaktadır. Ancak, bu algoritmalar genellikle iki problemin sıkıntısını çekmektedirler: yürütüm sürelerinin uzun olması veya çok fazla bellek gereksinimi. Önceden sunulan algoritmalardaki bu problemler, onların, gerçek zamanlı verilerle çalışan ve hızlı yanıt sürelerine ihtiyaç duyan dolaşım uygulamalarında kullanılmasını mümkün kılmamaktadır. Zamana bağımlı akış hızı modeli içeren en kısa yol seri algoritmasının, çok daha fazla bellek gerektirmeden, yürütüm süresinin hızlandırılması için "Değiştirilmiş Dijkstra" algoritmasını temel alarak paralel algoritmalar öneriyoruz. Cuda ve OpenMP kullanarak 3 farklı paralel gerçekleştirme geliştirdik: Bunlar (i) Cuda tabanlı uyarlama, (ii) OpenMP tabanlı uyarlama ve (iii) Cuda ve OpenMP tabanlı karma uyarlama. OpenMP uyarlamasında 10 kat, diğer uyarlamalarda da 17 kat hızlanma elde edilmiştir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

$A(i)$	Adjacency list of vertex i
d	Destination node
E	Set of edges of G
$\|E\|$	Number of edges in the edge set of E
$\|E_C\|$	Number of edges created
e_i	An edge from the set E
$\|E_R\|$	Number of edges left after unreachable vertices are removed
f_k	Start time of kth time interval
$f_{k_{ij}}$	Start time of kth time interval on edge (i, j)
G	Graph
(i, j)	Edge from vertex i to vertex j
k	Index of current time interval
K	Thousand ($= 10^3$)
l_{ij}	Length of edge (i, j)
$l_{ij}(t)$	Length of edge (i, j) at time t
$L_{reachables}$	List of reachable nodes
M	Million ($= 10^6$)
n	Number of nodes in G
$Path_{sd}(t)$	Shortest path from node s to node d starting at time t
$pred(i)$	Predecessor node of node i in the shortest path from s to i
s	Source node
S	Set of visited vertices
S_p	Speedup
t_i	Current minimum time to get from s to vertex i
T_s	Serial run time of an algorithm
T_p	Parallel run time of an algorithm
U	Set of unvisited vertices
V	Set of vertices of G

$\|V\|$	Number of vertices in the vertex set of V
$\|V_C\|$	Number of vertices created
v_i	A vertex from the set of V
v_{ij}	Velocity of a vehicle on edge (i, j)
$v_{k_{ij}}$	Flow-speed on edge (i, j) at kth time interval
$\|V_R\|$	Number of vertices left after unreachable ones are removed
w_{ij}	Weight of edge (i, j)
$w_{ij}(t)$	Weight of edge (i, j) at time t
ε	A Very Small Positive Number
η	Efficiency

LIST OF ACRONYMS/ABBREVIATIONS

CSR	Compressed Row Storage
DFS	Depth-First Search
DOT	Decreasing Order of Time
FIFO	First-In-First-Out
MD	Modified Dijkstra
TDG	Time Dependent Graphs
TDSPP	Time Dependent Shortest Path Problem

1. INTRODUCTION

In the last decade, with the advances in technology, navigation systems which helped people to get from point A to point B as fast as possible, has also changed greatly. A few years ago, most of the navigation devices were using only pre-installed maps to determine the route. These devices converted the map paths to graphs such that nodes are the destinations in the map and the edges are the paths between them. They were updating their graphs only if a road was added to or removed from the area they are handling. The shortest path route calculated using these old devices mostly did not change for a fixed pair of source and destination points and thus they were not very efficient since they were not taking into consideration the number of cars that were using those routes and causing lots of traffic jams.

In these days, state of the art navigation programs mostly benefit from (i) real-time data gathered through some devices installed on cars or trucks by companies that have an agreement with them or (ii) from camera data placed on mostly used route paths or (iii) from satellite data [1]. Hence, nowadays, navigation programs run on data which change in time. It is not enough for navigation programs to just use path information. They also have to take into consideration the other cars in their routes and traffic jams in areas where they travel. Thus, efficient solutions to the shortest path problem on time dependent graphs are required for navigation applications.

1.1. Related Work

A *time dependent graph (TDG)* is a graph in which its structure and parameters may change over time. Some of the previous works approach TDG as there are some nodes/edges additions/deletions in the graph, some other papers consider TDG so that the edge-delay, which is the total time spent to get from start of the edge to end of the edge in a given time, changes in time [2,3]. In this work, we take the model proposed by Sung *et al.* [4], where the speed of a vehicle travels changes in time intervals. This approach is more realistic one in the perspective of cars traveling in urban roads.

A commonly satisfied property in time dependent graphs is called the FIFO (First-In-First-Out) property which is also named as the non-passing property in the literature. This property states that if the same route from node *i* to node *j* is traveled then the one that leaves earlier from node i, arrives earlier at node j than the one that leaves later. This property is a realistic assumption that we also see in daily traffic. The FIFO property enables us to develop efficient and polynomial algorithms for time dependent shortest path problem (TDSPP) [5]. TDSPP is an NP-hard problem on TDG without the FIFO [6] property. In this work, we consider TDG in which the FIFO property holds.

The use of traditional graph algorithms are not designed for the time dependent graphs and hence they can not help us in new applications that employ dynamic graphs. In the last decades, there some research in the area of algorithms for TDG [4, 5, 7, 8] have been carried out. However, these new algorithms should run very efficiently on large graphs, since they are to be used in real-time devices that require fast response times. There can be thousands of new route queries in a minute in large cities. These devices should be able to answer these queries with the fastest route and also provide a time of arrival with a small range of error.

In the literature, there are some works on shortest path algorithms on TDGs that make the algorithm run more efficiently. Many of these works improve the running time of the algorithm but they enormously increase the memory usage which make them unsuitable for the devices. One of the methods used here is *time expansion method*. The algorithm first transforms TDG to a time independent graph and then solves the problem using traditional algorithms for static graphs. However, this method generates huge graphs and, in general, results non-polynomial algorithms run times [3, 7]. Also, these works, study on the TDSPP problem from one source node to a target node.

There are some works done in order to speed up the TDSPP by exploiting graph pre-processing before running the actual algorithm. Goldberg *et al.* [9], propose a landmark-based ALT algorithm on static graphs. This algorithm chooses a small number of landmarks and computes shortest paths between them in the pre-processing stage. It then utilizes this information to find lower bounds on the shortest paths from one source to a target node and reduces the total number of nodes to be visited in order to reach the destination node. Ohshima *et al.* [8], presents a modified version of this algorithm which runs on dynamic graphs. With the help of some preprocessing, they achieve up to 4-fold speed up when compared to the Modified Dijkstra algorithm that was first proposed by Dreyfus [10] to solve the shortest path problem on time dependent graphs. This algorithm also work on TDSPP as defined from one source node to one destination node in the graph.

A lot of work appear on the parallelization of algorithms for shortest path problem on graphs which are not time dependent [11]. There are also some works which provide parallel programming in TDSPP. These works consider the TDG model in which the edge-lengths are time dependent [12–14]. However, we are not aware of any previous works that considers parallelization of the shortest path problem from a source node to all other nodes in time dependent graphs, which use flow-speed model.

1.2. Contributions of the Thesis

In this thesis, we consider the problem of TDSPP from a single source node to all other nodes in the graph. In real life, navigation applications can utilize our parallel algorithms to calculate shortest paths from central locations of cities to all other places beforehand so that people searching for these paths can be informed about latest travel times.

We use the time dependent flow-speed model defined in [4]. We develop three parallel algorithms for the shortest path problem defined in this model: (i) an OpenMP version, (ii) a Cuda version for GPUs and (iii) a hybrid Cuda and OpenMP hybrid implementation. To the best of our knowledge, this is the first study of GPU-based parallelization work on the time dependent shortest path problem. We also analyze these algorithms with respect to the performance metrics defined for parallel computer systems.

In Chapter 2, we review the background material on graphs, shortest path algorithms and parallel processing performance metrics. Chapter 3 presents the background work on time dependent graphs and algorithms. Chapter 4 presents the three parallel algorithms we developed for time dependent graphs. In this chapter, we also present the data structures used. Chapter 5 presents the performance results of our algorithms based on tests carried on several graphs generated using the Graph500 reference code [15]. Finally, Chapter 6 concludes the thesis with a discussion of results and possible future work.

2. PRELIMINARIES

This chapter provides preliminary information on graphs, traditional shortest path problems and parallel programming performance metrics.

2.1. Graphs

Let $G = (V, E)$ be a *directed graph*, where $V = \{v_1, v_2, ..., v_n\}$ is a set of vertices and $E = \{e_1, e_2, ..., e_m\} \subseteq V \times V$ is a set of *directed* edges. Here, $e_i = (u, v)$ means that u is the beginning vertex and v is the ending vertex of the edge e_i . Note that, (u, v) and (v, u) represent distinct edges of G and having $(u, v) \in E$ does not imply that $(v, u) \in E$ and vice versa. Also, more than one edge having the same beginning and ending vertices is not allowed in directed graphs. Let l_{ij} be the length of an edge from vertex i to vertex j. We also let $A(i)$ denote the adjacency list of vertex i, where each element of $A(i)$ is the ending vertices of edges coming out of vertex i. Therefore, $j \in A(i) \iff (i, j) \in E$. Throughout the thesis, the followings terms are used interchangeably:

- vertex, node
- edge, arc
- edge length, edge weight
- adjacency list, neighbour list

2.1.1. Road Map Graphs

Road map graphs are utilized while solving real life problems such as finding a path from one city to another. In road map graphs vertices corresponds to intersections and edges correspond to road segments.

In road map graphs, there is a notion of a car traveling through the map. Let v_{ij} be the fixed velocity of the car on the edge (i, j). Then, the travel time of the car to pass an edge $(i, j) \in E$ is calculated as l_{ij}/v_{ij}.

2.1.2. Depth-First Traversal

Depth-first search(DFS) is used to traverse or search a graph. The algorithm starts from a given node (or root when used in tree) and tries to go as much as possible in a branch so that it returns to another branch after all the nodes visited in that branch.

The recursive algorithm of DFS is given in [16]. Pseudocode of the algorithm is presented in the Figure 2.1.

```
Function DFS
 1: Inputs: Graph $G$, node $v$
 2: Goal: To visit all nodes that can be reached in graph
 3: Output: $L_{reachables}$, list of all nodes in $G$ which are reachable from node $v$
 4: set $v$ as visited, and put node $v$ to $L_{reachables}$ list
 5: for each node $w$ in neighbor list of node $v$ do
 6:   if $w$ is not labeled as visited then
 7:     call Function DFS$(G, w)$ recursively.
 8:   end if
 9: end for
10: return $L_{reachables}$
```

Figure 2.1. Recursive Depth-First Search Algorithm.

Example of a running DFS algorithm is presented in Figure 2.2. In the figure, straight lines are representing edges between nodes in the graph. Dotted lines shows the nodes visited in the order of DFS algorithm visits them. Algorithm starts from node A and the result of the algorithm is $L_{reachables} = A, B, E, F, G, C, D$.

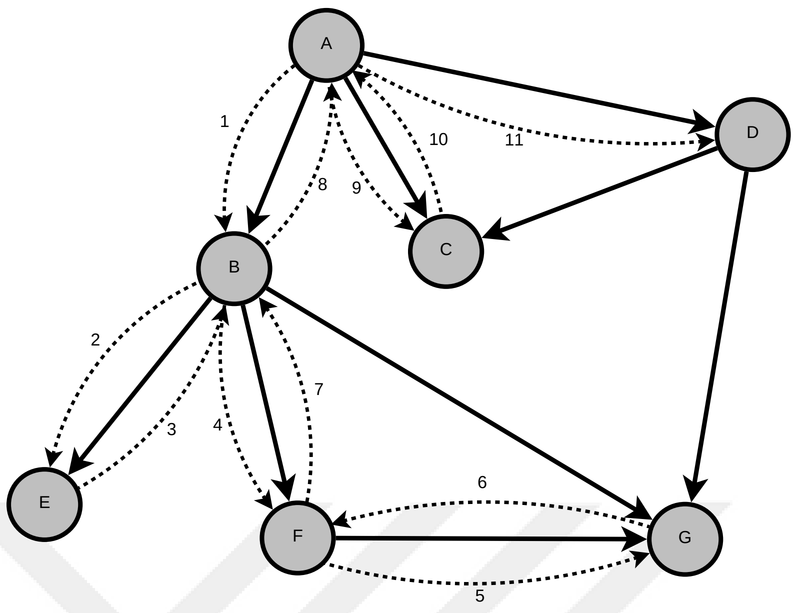


Figure 2.2. A running example of Depth-First Search Algorithm.

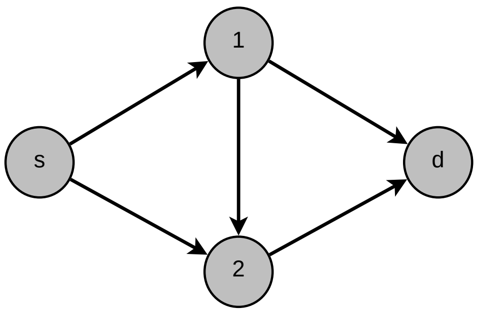
2.1.3. Time Dependent Graphs

Let $G = (V, E)$ be a *time dependent graph (TDG)*, where the elements in G changes in time. This change could be in one these forms: removal of an existing vertex from V or addition of a new vertex to the set of V. The same actions can be taken on edges.

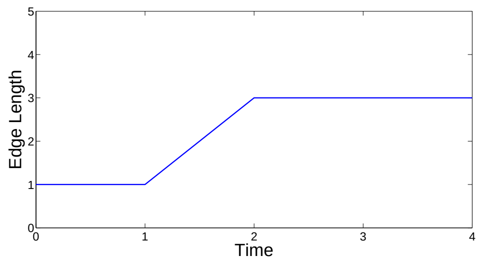
Another approach considered in studying time dependent graphs is to think of edge lengths as a time dependent variable. In this approach, length of an edge is represented as $l_{ij}(t)$, where i is the starting node of edge, j is the ending node of edge and t is the time. This method is used widely [2, 3, 8, 10, 17].

An example TDG with edge lengths that change with respect to time is shown in Figure 2.3. This figure consists of following 6 figures:

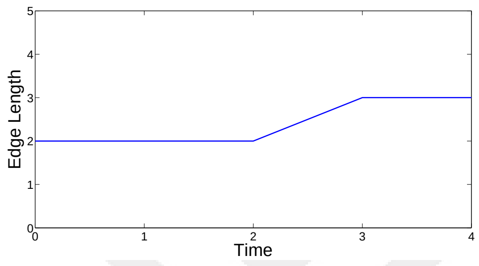
(a) a TDG, $G(V, E)$, with edge lengths that change with respect to the time;
(b) Length of edge $(s, 1)$ at time t, $l_{s1}(t)$,
(c) Length of edge $(s, 2)$ at time t, $l_{s2}(t)$,
(d) Length of edge $(1, 2)$ at time t, $l_{12}(t)$,



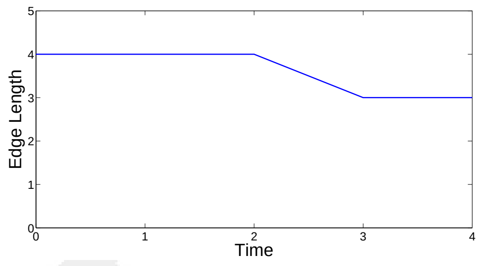
(a) A TDG, $G(V, E)$.



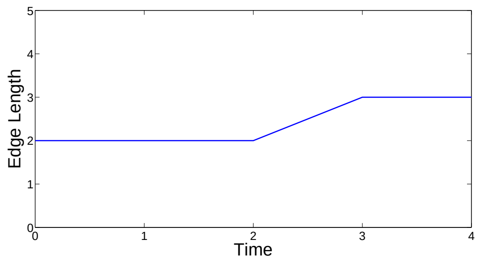
(b) Length of edge $(s, 1)$ at time t.



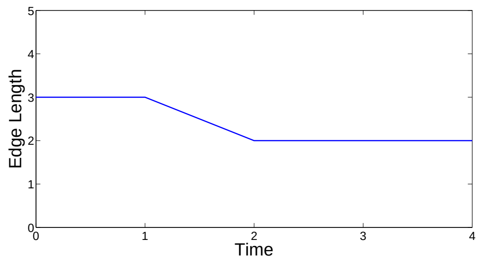
(c) Length of edge $(s, 2)$ at time t.



(d) Length of edge $(1, 2)$ at time t.



(e) Length of edge $(1, d)$ at time t.



(f) Length of edge $(2, d)$ at time t.

Figure 2.3. A TDG and its time dependent edge lengths vary over time.

(e) Length of edge $(1, d)$ at time t, $l_{1d}(t)$,

(f) Length of edge $(2, d)$ at time t, $l_{2d}(t)$.

In the graph depicted in Figure 2.3, the shortest path from node s to node d is the route of $s - 1 - d$ if the starting time t is 0 and the vehicle's speed is always 1. The time that elapses while going from node s to node d is calculated as $Path_{sd}(t = 0) = l_{s1}(t = 0) + l_{1d}(t = t_1)$, where t_1 is the time taken from node s to node 1. Then, $Path_{sd}(t = 0) = l_{s1}(0) + l_{1d}(1) = 1 + 2 = 3$.

Sung *et al.* [4] propose another model to time dependent graphs. This model considers flow speeds of arcs as time dependent parameters. The velocity of a car traveling through the edge (i, j) at time t is denoted by $v_{ij}(t)$. This model is analyzed in Section 3.3.

2.1.4. FIFO Property

FIFO property, also called non-passing property, says that if two cars (a and b) travel from vertex i to vertex j, the one (e.g car a) which departs from i earlier than the other (e.g car b) arrives at j earlier than the other (i.e. car b). A time dependent graph ensures FIFO property, if all the edges in the graph satisfy the FIFO property. We can formulate this as follows: Let $w_{ij}(t)$ be the traveling time on edge (i, j) at time t. The arc (i, j) has FIFO property if:

$$t_1 < t_2 \implies t_1 + w_{ij}(t_1) < t_2 + w_{ij}(t_2) \text{ for all } 0 < t_1 < t_2.$$

Orda *et al.* [18] shows that, FIFO property can be achieved in non-FIFO networks by allowing waiting at the vertices. This can be achieved by defining minimum waiting times on starting vertices of edges which break the FIFO property. However, this is an expensive process as we need examine all time intervals to decide the minimum waiting time for a vertex. Also, waiting times in vertices are not preferred by most of the applications. On the other hand, FIFO property is a reasonable assumption in the case of real life traffic routes. Because in general, we exit a path after the cars which entered this path before us.

Kaufman *et al.* [19] and Orda *et al.* [18] show that the shortest path problem in time dependent networks is polynomially solvable. Sherali *et al.* [6] proves that when the FIFO property is not ensured, the problem of finding shortest path becomes NP-hard.

Orda *et al.* [18] gave an example of a simple non-fifo graph where the shortest path includes infinite number of edges. The example graph is presented in Figure 2.4, where the length of the edges are defined as follows:

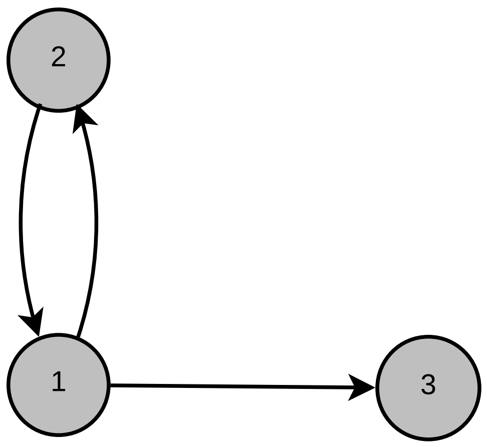


Figure 2.4. An example of a time dependent graph without FIFO property.

$$l_{12}(t) = l_{21}(t) = \begin{cases} \dfrac{1-t}{2} & 0 \leq t < 1, \\ 1 & \text{otherwise.} \end{cases}, \; l_{13}(t) = \begin{cases} 3-2t, & 0 \leq t < 1, \\ 1, & \text{otherwise.} \end{cases}$$

When we examine to the shortest path from vertex 1 to vertex 3 at time $t = 0$, the resulting shortest path is $(1, 2, 1, 2, 1, 2, 1, ..., 1, 2, ..., 3)$, even if the shortest path length is 2 the path includes infinite number of edges.

2.2. Shortest Path Problem

Shortest path problem is stated as finding a path from a source vertex s to a destination vertex d in a graph so that sum of the lengths of edges in that path is minimum among other paths from s to d. Let $P = \{v_i, v_, ..., v_j\}$ be a path from node v_i to node v_j, such that every consecutive pair of vertices in the path is in E $((v_k, v_{k+1}) \in E$ for $i \leq k \leq j)$. With respect to the path P, the distance from v_i to v_j is calculated by $\sum_{k=i}^{j} l_{v_k v_{k+1}}$. Shortest path problems can be classified into the following four groups:

(i) One-to-One,
(ii) One-to-All,
(iii) All-to-One,
(iv) All-to-All (All pairs).

2.2.1. One-to-One

Shortest path problem from one source node to a target node. All of the minimum arrival times from the source node to other nodes in graph are not necessarily calculated.

One technique to the increase the efficiency of the algorithm is to reduce the search space of the algorithm. Goldberg *et al.* [9] presented a landmark based ALT algorithm which basically diminishes the number of visited vertices for getting from the source node to the destination node. ALT algorithm based on choosing pre-calculated landmarks to define a temporary distance labels for nodes. Performance of the algorithm depends on the landmark selection. Both of the papers [8,9] describe some ways to choose landmarks properly.

2.2.2. One-to-All

Shortest path problem from one source node to all other nodes in the graph. Solving one-to-all problem results a *shortest-path tree* in which the source node is the root of the tree and the path distance from root to another node in tree is the shortest path distance in graph.

Dijkstra's algorithm [20], which is explained in Section 2.2.5, is the most famous algorithm to solve One-to-All shortest path problem. Note that, the original Dijkstra's algorithm finds shortest path between two nodes, but generally it is used to find one-to-all shortest paths in graphs.

Bellman-Ford(also named Bellman-Ford-Moore [21]) algorithm is another well-known algorithm which solves shortest path problem from a single source node to all other nodes in graph [22–24]. This algorithm is slower than Dijkstra's algorithm but it has the ability to solve this problem in graphs with negative weights.

The one-to-all shortest path problem in time dependent graphs is the main focus of this thesis as it is the subject of solving in parallel programming methodology. Modified Dijkstra algorithm is used to solve this problem which explained later in Section 3.4.

2.2.3. All-to-One

Shortest path problem from all nodes to a target node in the graph. This problem is useful when considering a real life case in which people from all around the city try to reach the most centralized part of the city. If those people use navigation devices or applications, the application should be able to calculate all-to-one shortest path problem in order to respond quickly to the queries coming from users.

Chabini *et al.* [25, 26] proposed the DOT algorithm to all-to-one fastest paths problem for all departure time intervals in time dependent graphs. The algorithm is proved to has the optimal running time which is the complexity of the problem [25–27].

2.2.4. All-to-All (All Pairs)

This problem calculates minimum arrival times from a node to all other nodes for each node. So, minimum arrival times calculated for all-pairs in the graph. The well known Floyd-Warshall algorithm [28] solves the all-pairs shortest path problem in a positive or negative weighted graph, though it does not reveal the shortest paths' routes. Run time complexity of this algorithm is $\mathcal{O}(|V|^3)$, where $|V|$ is the number of vertices in graph.

Pettie *et al.* [29] proposed a new algorithm to all-pairs shortest path with real-weighted graphs. The method makes an approximation to minimum arrival times and make use of this information when calculating the real values. This algorithm runs in $\mathcal{O}(|E||V| + |V|^2 loglog|V|)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph.

2.2.5. Dijkstra's Algorithm

Dijkstra's algorithm [20] is an algorithm which finds shortest paths between nodes in a graph. The run time complexity of the algorithm is $\mathcal{O}(|V|^2)$.

```
Algorithm Dijsktra
1: Initialize
2: $S \leftarrow \emptyset$
3: $U \leftarrow V$
4: $t_i = \infty$ for each vertex $i \in V$
5: $t_s = 0$
6: $pred(s) = 0$
7: while $|S| < n$ do
8:     let $i \in U$ be a vertex for which $t_i = \min\{t_j : j \in U\}$
9:     $S \leftarrow S \cup \{i\}$
10:    $U \leftarrow U - \{i\}$
11:    for each$(i, j) \in A(i)$, where $j \in U$ do
12:        $alt \leftarrow t_i + l_{ij}$
13:        if $t_j > alt$ then
14:            $t_j \leftarrow alt$
15:            $pred(j) \leftarrow i$
16:        end if
17:    end for
18: end while
19: return $t[\,]$ and $pred[\,]$
```

Figure 2.5. Dijkstra Algorithm.

In the pseudocode of the algorithm given in Figure 2.5 the following notation is used:

- S is the set of visited vertices, so far, in graph
- U is the set of unvisited vertices, so far, in graph

- s is the source node
- V is the set of vertices in graph
- n is the number of vertices in graph
- t_i is the current minimum distance of node i to the source node s
- $pred(i)$ is the predecessor vertex of node i in the shortest path route from s to i
- $A(i)$ is the list of adjacencies of node i
- alt is the alternative route time that is tested for the fastest route from vertex s to j

2.3. Parallel Programming Performance Metrics

The following definitions are taken from [30–32].

Serial run time is the time elapsed between the beginning and the end of the program on a sequential computer. It is denoted by T_s. *Parallel run time* of an algorithm is the time passed from the beginning of parallel execution to the end of last processing component of the parallel implementation. It is denoted by T_p.

Speedup(S_p) is the ratio of time taken to run the sequential program on a single processing unit to the time taken to run the parallel program on a parallel computer. In an ideal system, S_p should be equal to the number of processors running on the computer. It is formulated as:

$$S_p = \frac{serial\ execution\ time}{parallel\ execution\ time} = \frac{T_s}{T_p}$$

Efficiency(η) is another metric which is closely related to the speedup metric. It is defined as the ratio of speedup to the number of processors used in parallel execution. In an ideal system, η should be equal to 1.

$$\eta = \frac{S_p}{p} = \frac{T_s}{pT_p}$$

Scalability is used to address the change in the performance of the program as the problem size and the number of processing units increase. Scalability measures the program run time performance with respect to increasing workloads and the number processing units at the same rate.

3. SHORTEST PATH PROBLEM ON TIME DEPENDENT GRAPHS

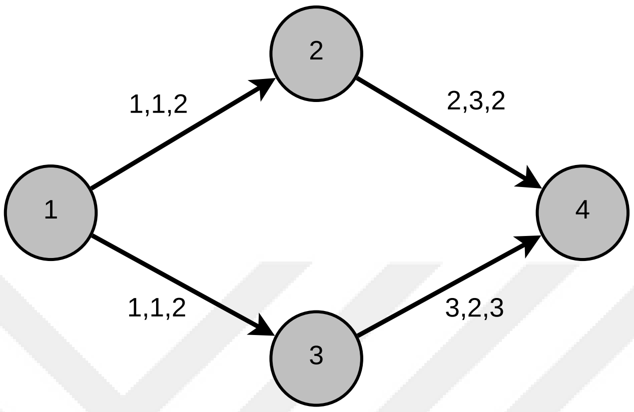
One of the most studied problem on time dependent graphs is the shortest path problem. In this chapter, we review the previous work that appears in the literature. The problem that is mostly studied on TDGs involve graphs in which lengths of edges are time dependent.

3.1. Shortest Path Algorithms on TDG with Varying Edge Lengths

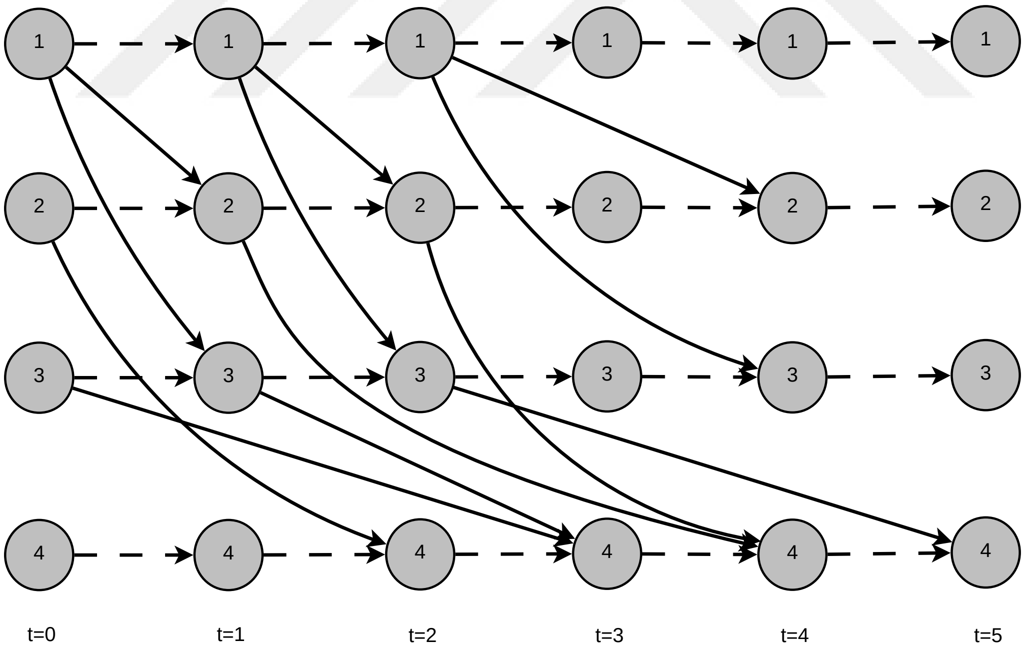
One of the proposed methods for solving TDSPP is to use time expanded graphs. This method converts the time dependent graph to a static graph. It creates a copy of each element of TDG for each time instance. This results in very large static graphs. When the number of time intervals increase, this method needs to create a huge graph and the algorithm run time can become non-polynomial [8]. Time expansion method is often used where the FIFO property does not hold. This model converts a time dependent graph to a static graph where we can use other algorithms used in static graphs.

In the Figure 3.1, an example of time-expanded graph is shown. The upper graph, $G(V, E)$, at the figure is a TDG with edge delays vary in time, the edge delays are placed at the edges for time $t = 0, 1, 2$. The bottom graph, $G_E(V_E, E_E)$, is the time-expanded graph of $G(V, E)$. For each time period, nodes and edges in $G(V, E)$ are duplicated for graph $G_E(V_E, E_E)$. Dotted lines in $G_E(V_E, E_E)$ connects the duplicate nodes created for each time period. In this figure, it can be seen that converting $G(V, E)$ to $G_E(V_E, E_E)$ increases graph size enormously, but resulting expanded graph G_E is a time independent graph.

Some research has been done in order to speed up the TDSPP by exploiting pre-processing of the graph before running the algorithm. Goldberg *et al.* [9], proposes



(a) A TDG, $G(V, E)$, with edge delays vary in time.



(b) Expanded graph $G_E(V_E, E_E)$ of graph $G(V, E)$.

Figure 3.1. An illustration of time-expanded graph generation.

landmark-based ALT algorithm on static graphs. This algorithm chooses a small number of landmarks and computes shortest paths among them as part of preprocessing. It, then, utilizes this information to find lower bounds to the problem of shortest path from one source to a target node and reduces the total number of nodes to visit in order to reach the destination node. Ohshima *et al.* [8], present a modified version of this algorithm which runs on dynamic graphs. With the help of some preprocessing, they achieve up to 4 times speed up with respect to the Modified Dijkstra algorithm which was first proposed by Dreyfus [10] in order to solve shortest path problem on TDGs. This algorithms also work for one source node to one destination node TDSPP on the graph. They assume graphs with time-dependent edge lengths and FIFO property.

Ding *et al.* [2] and Kanoulas *et al.* [17] study shortest path problem on time dependent graphs for a given starting interval from the source node. That is, they examine the least total travel time from a source node v_s to a destination node v_d with the departure time from v_s given as a time interval input. They find the best departure time which results in the shortest time to get from v_s to v_d. Note that waiting time in v_s is not considered in the time calculation. Kanoulas *et al.* [17] propose an algorithm which is an extension of A* algorithm. Their approach is to expand the graph by using expanded nodes set and a priority queue list which at first consists only of the source node and then expanded by taking neighbors of the element in priority queue. Ding *et al.* [2] propose a Dijkstra based algorithm for FIFO and non-FIFO graphs. The algorithm consists of two steps: time refinement and path selection respectively. In the first step, it calculates earliest arrival times for all nodes when departing from source node at any time in the given interval. The second step chooses the optimal path with an optimal starting time.

Chabini *et al.* [25, 26] proposes Decreasing Order of Time (DOT) algorithm for all-to-one fastest paths problem for all departure times and proves that it has the optimal worst-case running time complexity for this problem. They assume the edge lengths are time dependent variables but will be static after a finite number of time intervals M. Let $S = \{t_0, t_1, ..., t_{M-1}\}$ be time intervals. Then the problem becomes a static shortest path problem when the departure time is greater than t_{M-1}. Their method calculates fastest arrival time with static shortest path problem for time t_{M-1}. Then, they calculate the fastest paths starting from time t_{M-2} to time t_0. DOT name comes from this approach. The running complexity of DOT algorithm is $\mathcal{O}(SPP + |V|M + |E|M)$ [26], where SPP is the static shortest path problem, $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph.

3.2. Parallel Algorithms

Chabini and Ganugapati [12] propose parallel algorithms based for the DOT algorithm given by [25, 26]. Because the DOT algorithm has the optimal run time complexity for the problem of all-to-one fastest paths for all departure times, they try to improve the performance of this algorithm by investigating parallel implementations of DOT algorithm. They implement distributed and shared memory parallel implementations that achieve approximately 4 fold speedup with 6 processors on input graphs with 1000 nodes, 3000 edges and 100 time intervals [12].

Tremblay *et al.* [13] also proposes parallelization works for the DOT algorithm and the time dependent least time path algorithm given by [33]. They achieve up to 13 fold speedup on 15 processors [13]. Also, Ziliaskopoulos *et al.* [14] examine parallel designs for the algorithm given in [33]. They improve the algorithm run time by achieving 2 fold speedup on shared memory implementation with 4 CPUs, and achieve 7 fold speedup with message passing algorithm using parallel virtual machine (PVM). In theses parallel algorithms, they consider the cases where the time varying travel times for edges will be constant after a known time interval.

3.3. Flow-Speed Model

Flow-Speed Model is introduced by Sung *et al.* in [4]. The main idea that distinguishes this model from the other models used on TDGs is that the flow speed on each edge depends on the time interval instead of time dependent edge lengths that is considered in most of the TDG problems in previous works.

Let $G = (V, E)$ be a graph, where V is the set of vertices and E is the set of edges and l_{ij} is the non-negative length of the edge (i, j). Consider dividing the time horizon into K time intervals in which an interval represented by $[f_k, f_{k+1})$, where $k = 0, 1, ..., K - 1$ and $0 \leq f_0 < f_1 < ... < f_{k-2} < f_{k-1}$. Let $v_{k_{(i,j)}}$ be the non-negative flow speed on the edge (i, j) in the time interval $[f_k, f_{k+1})$. For each vertex $j \in V$, define a value $t_j = min_{i \neq j}\{T(t_i, (i, j))\}$, where $T(t_i, (i, j))$ is the travel time from vertex i to j starting at time t_i. Then, the purpose of this model becomes finding values of t_j for all vertices $j \in V$, where the starting time from the starting node given as t_s.

Theorem 3.1. *Assume that two vehicles departing from vertex s at times t_1 and t_2 and they arrive the vertex d at time $T(t_1)$ and $T(t_2)$ respectively. Then, the following must be provided in flow-speed model [4]:*

$$\text{if } t_1 \leq t_2 \implies T(t_1) \leq T(t_2).$$

Proof. [4]: Let us assume that $t_1 \in [f_i, f_{i+1})$, $t_2 \in [f_j, f_{j+1})$, $T(t_1) \in [f_k, f_{k+1})$ and $T(t_2) \in [f_m, f_{m+1})$, where $i \leq j \leq k, m$, and the length of the arc (s, d) is $l_{s,d}$. Then, consider the length of edge (s, d) as a summation of the interval times multiplied by velocities of the vehicle at that interval until the vehicle reaches the node d:

$$\begin{aligned} l_{s,d} &= v_{m_{(i,j)}}(T(t_2) - f_m) + v_{m-1_{(i,j)}}(f_m - f_{m-1}) + ... + v_j(f_{j+1} - t_2) \\ &= v_{k_{(i,j)}}(T(t_1) - f_k) + v_{k-1_{(i,j)}}(f_k - f_{k-1}) + ... + v_i(f_{i+1} - t_1) \\ &\geq v_{k_{(i,j)}}(T(t_1) - f_k) + v_{k-1_{(i,j)}}(f_k - f_{k-1}) + ... + v_j(f_{j+1} - t_2), \end{aligned}$$

since $i \leq j$ and $t_1 \leq t_2$. This means

$$v_{m_{(i,j)}}(T(t_2) - f_m) \geq v_{k_{(i,j)}}(T(t_1) - f_k),$$

where $k = m$, or

$$v_{m_{(i,j)}}(T(t_2) - f_m) + v_{m-1_{(i,j)}}(f_m - f_{m-1}) + ... + v_{k_{(i,j)}}(f_{k+1} - f_k) \geq v_{k_{(i,j)}}(T(t_1) - f_k),$$

where $k < m$, or

$$v_{m_{(i,j)}}(T(t_2) - f_m) \geq v_{k_{(i,j)}}(T(t_1) - f_k) + v_{k-1_{(i,j)}}(f_k - f_{k-1}) + ... + v_{m_{(i,j)}}(f_{m+1} - f_m),$$

where $k > m$. The third inequality function contradicts with the assumption of $v_{k_{(i,j)}} \geq 0$ because $T(t_2) < f_{m+1}$, then $k \leq m$. Therefore, we conclude that $T(t_1) \leq T(t_2)$. $\square$

Theorem 3.1 proves that flow-speed model ensures the FIFO property.

3.4. Modified Dijkstra Algorithm

The algorithm that we work on is similar to Dijkstra based algorithm which was first proposed by Dreyfus *et al.* in [10] for edge-length varying TDGs. Sung *et al.* [4] present Modified Dijkstra (MD) algorithm for solving problems using the flow-speed model on time dependent graphs. They take the complexity of Dijkstra's algorithm as $\mathcal{O}(|V|^2 + |E|)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges in graph. MD algorithm's run time complexity is $\mathcal{O}(|V|^2 + |E|K)$ [4], where K is the maximum number of time intervals scanned in ArrivalTime function that is presented in Figure 3.3.

There are two differences between MD algorithm and Dijkstra's algorithm. The first difference is that updating the neighbours of the node with the smallest arrival time amongst unvisited nodes differs in these two algorithms. In MD algorithm, there could be an update for each neighbour of that node without taking into consideration whether that neighbour is visited yet or not. On the other hand, Dijkstra's algorithm only updates unvisited neighbours. The second difference between these two algorithms is calculating the time passed to get from the node which has the smallest arrival time amongst the unvisited nodes to a neighbour of that node. This difference is because of the fact that MD algorithm deals with TDGs whereas Dijkstra's algorithm solves the shortest path problem for static graphs. *ArrivalTime* function in the MD algorithm calculates the arrival time to a neighbour by considering the current time.

```
Algorithm Modified Dijkstra
1: $S \leftarrow \emptyset; U \leftarrow V$
2: $t_i = \infty$ for each vertex $i \in V$
3: $t_s = 0$ and $pred(s) = 0$
4: while $|S| < n$ do
5:     let $i \in U$ be a vertex for which $t_i = \min\{t_j : j \in U\}$
6:     $S \leftarrow S \cup \{i\}$
7:     $U \leftarrow U - \{i\}$
8:     for each$(i, j) \in A(i)$ do
9:         if $t_j > ArrivalTime(t_i, (i, j))$ then
10:            $t_j = ArrivalTime(t_i, (i, j))$ and $pred(j) \leftarrow i$
11:        end if
12:    end for
13: end while
```

Figure 3.2. Modified Dijkstra Algorithm.

```
ArrivalTime$(t_i, (i, j))$ Function
1: $Arclength \leftarrow l_{ij}$
2: let $k \in \{0, 1, 2, ..., K\}$ be an index for which $f_{k_{(i,j)}} \leq t_i < f_{k+1_{(i,j)}}$
3: $Arclength \leftarrow Arclength - v_{k_{(i,j)}} \times (f_{k+1_{(i,j)}} - t_i)$
4: while $Arclength > 0$ do
5:     $k \leftarrow k + 1$
6:     $Arclength \leftarrow Arclength - v_{k_{(i,j)}} \times (f_{k+1_{(i,j)}} - f_{k_{(i,j)}})$
7: end while
8: $ArrivalTime(t_i, (i, j)) \leftarrow f_{k+1_{(i,j)}} + Arclength/v_{k_{(i,j)}}$
9: return
```

Figure 3.3. Function for calculating arrival time.

4. PARALLEL ALGORITHMS FOR SHORTEST PATH PROBLEM ON TIME DEPENDENT GRAPHS

The major challenge of the shortest path problem on TDG is that the solution requires a lot of time and memory. To address this challenge, we resort to parallelization of the algorithm for multi-core and GPU systems. Parallel programming is carried out using OpenMP and Cuda. In this work, MD algorithm is taken as a sequential solution to the shortest path problem on TDG and parallelized.

4.1. Time Consuming Parts of Sequential Modified Dijkstra Algorithm

First, we implement a sequential version of MD algorithm in order to detect most time consuming parts of the algorithm. We use a CPU profiler program called `Very Sleepy` [34] to analyze the sequential implementation of the MD algorithm. `Very Sleepy` program shows the current working processes in the computer as a list. We implement and run sequential MD algorithm on Visual Studio and in the mean time, find our process on the list provided by `Very Sleepy` program. After double-clicking on the process, the program starts to collect data about the running process. When the process finishes, `Very Sleepy` shows all the function calls in the order of decreasing total elapsed time in the functions.

In the MD algorithm, we observe that there are mainly two parts of this algorithm which consumes most of the run time of the program and which can be parallelized without causing too much overhead. The analysis results show that the most time consuming part of this algorithm is finding an unvisited node with the smallest arrival time. This is shown on the 5th line of the MD algorithm pseudo code given in Figure 3.2. Another time consuming part is where we update minimum arrival times of every neighbour of the node with the smallest arrival time amongst the unvisited nodes and calculate the arrival times of these neighbours while updating. This part is the for loop on lines 8-12 in Figure 3.2.

4.2. Graph Data Structures

When working with large graphs, the algorithms require large computer memory and run times. In our implementation, we try to reduce the memory usage by using Compressed row Storage (CSR) format for storing our graph and its related properties. CSR format is widely used for storage of large graphs and sparse matrices. An example of CSR format storage is given at Figure 4.1.

Figure 4.2 shows the members of the Graph class and the node structure that are used to implement the data structures in C++. In the Graph class, *nodes* vector stores the information of each node. *xadj* vector holds the number of edges coming out from each node respectively to the *nodes* vector order. Similarly, *adjncy* vector holds the end points of the edges for each node respectively. Lastly, *adjwgt* vector holds the weights (lengths) of those edges.

In order to reduce the running time of the algorithms, we do not implement a node as a *class*. In this way, we can get rid of the constructor and destructor calls and other unnecessary time consuming overheads because of using the *class* structure. We represent a node as a structure. This also saves program memory.

In our shortest path algorithms, we calculate shortest paths from a *source* node to all other nodes in the graph. We first remove unreachable nodes from the *source* node in the graph before running any algorithm to find shortest paths. We use depth-first search to find the reachable nodes from the start node, and remove the unreachable ones. The time taken to find the reachable nodes is not included in the algorithm's reported run times because this is done only once before the algorithm starts and all of the algorithms can be run after this step. This is a small time to be concerned about and since it has nothing to do with the shortest path algorithm's run time performance, we do not include it in the reported results.

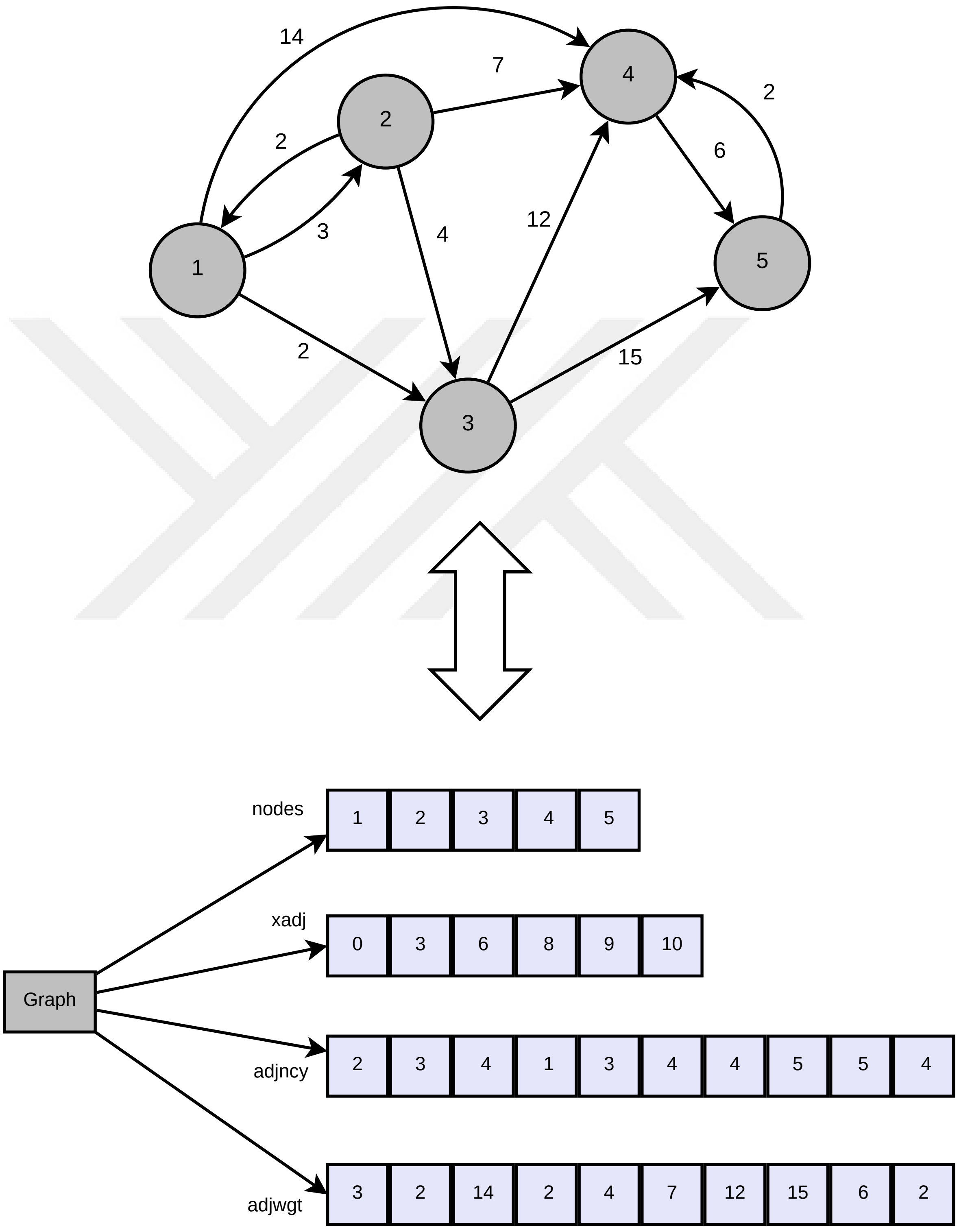


Figure 4.1. An example of CSR format representation of a graph.

```
class Graph{
    vector<Node> nodes;
    vector<int> xadj;
    vector<int> adjncy;
    vector<double> adjwgt;
    thrust::device_vector<Node> nodesD;   //only in cuda implementations
}

struct Node {
    int     nodeId;
    int     predecessorNodeId;
    int     minArrTime;
    bool    isVisited;
    int     position; //insertion order number
}
```

Figure 4.2. Graph class and Node structure.

4.3. OpenMP Based Implementation

In our OpenMP implementation, we parallelize (*i*) the finding of the unvisited node with minimum distance and (*ii*) the updating of minimum arrival times of the neighbours of that node. When we update the minimum arrival time for a node, we compare the current arrival time of the updated node with the new calculated arrival time. New arrival time is the sum of current time and the time passed to get from current minimum distanced node to neighbour node. Therefore, updating of the arrival times of each neighbour are independent from each other. Hence, we can safely parallelize this part without considering any race condition between threads.

In Figure 4.4, an example given which demonstrates how distances of neighbors of the current node updated with different threads. In this figure, i represents the node we found in $FindMinArrTimedUnvisitedNode\_OpenMP$ function, and $1, 2, 3, 4, 5, 6, 7, 8$ are represents the neighbors of that node. Let say we have 3 threads, namely T_1, T_2, T_3 and assume that updating a node's minimum arrival time take the same time for each

Algorithm OpenMP Modified Dijsktra

```
1: $S \leftarrow \emptyset; U \leftarrow V$
2: $t_i = \infty$ for each vertex $i \in V$
3: $t_s = 0$ and $pred(s) = 0$
4: while $|S| < n$ do
5:    i = FindMinArrTimedUnvisitedNode_OpenMP
6:    $S \leftarrow S \cup \{i\}$
7:    $U \leftarrow U - \{i\}$
      #pragma omp parallel shared{$node_i$} private{$node_j$}
8:    for each$(i, j) \in A(i)$ do
9:       if $t_j > ArrivalTime(t_i, (i, j))$ then
10:         $t_j = ArrivalTime(t_i, (i, j))$ and $pred(j) \leftarrow i$
11:      end if
12:   end for
13: end while
```

Figure 4.3. OpenMP implementation of Modified Dijkstra Algorithm.

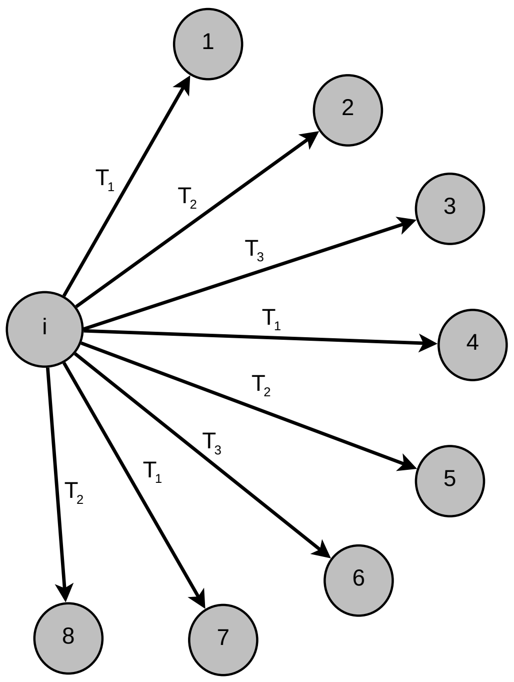


Figure 4.4. Example of minimum distance update in OpenMP implementation.

```
Function FindMinArrTimedUnvisitedNode_OpenMP

1: int $nodeId, t\_nodeId$
2: double $minArr, t\_minArr$
3: $nodeId \leftarrow -1, minArr \leftarrow MAX\_DISTANCE$
4: #pragma omp parallel reduction(+:count)
   shared{$minArr, nodeId$} private{$t\_nodeId, t\_minArr$}
5: $t\_nodeId \leftarrow nodeId, t\_minArr \leftarrow minArr$
6: #pragma omp for
7: for each $i \in U$ do
8:     $count \leftarrow count + 1$
9:     if $i.minArrTime < t\_minArr$ then
10:        $t\_minArr \leftarrow i.minArrTime$ and $t\_nodeId \leftarrow i.nodeId$
11:    end if
12: end for
13: #pragma omp critical
14: if $t\_minArr < minArr$ then
15:     $minArr \leftarrow t\_minArr$ and $nodeId \leftarrow t\_nodeId$
16: end if

17: return $i$ where $i.nodeId$ equals $nodeId$
```

Figure 4.5. Function for finding an unvisited node with minimum arrival time using OpenMP.

thread. Then, T_1 updates minimum arrival times for nodes $1, 4, 7$ by calculating passed times to go through edges $(i, 1), (i, 4), (i, 7)$; T_2 updates nodes $2, 5, 8$ with calculating elapsed times on edges $(i, 2), (i, 5), (i, 8)$; and T_3 updates arrival times of nodes $3, 6$ using edges $(i, 3), (i, 6)$ respectively. Note that, when considering updates of minimum arrival times of neighbours for a node, update of an edge does not have any influence over the other edges because the distance calculation of each one is independent from the others.

Finding of the minimum distanced unvisited node is done by using the parallel reduction operator in OpenMP. After the reduction, we apply a critical region where we only compare the arrival times gathered by each processor. This critical region does not cause a considerable problem in program running time.

4.4. Cuda/Thrust Based Implementation on a GPU

In our Cuda implementation also, we try to minimize the time where we find the element with minimum distance from among all the unvisited elements (i.e. line 5 in Modified Dijkstra's Algorithm in Figure 3.2). Because it is where the most of the running time is consumed, and it is appropriate to run it in parallel on a GPU. We used $thrust :: min\_element$ function from the Cuda/Thrust library to find the element with minimum distance in a $thrust :: device\_vector < Node >$. We need to define a binary predicate that is used for comparison by Thrust. Here, we consider not only the minimum distance criterion for the element but also whether the element has been visited or not. This predicate is shown in the Figure 4.8.

```
Algorithm Cuda Modified Dijsktra
 1: $S \leftarrow \emptyset; U \leftarrow V$
 2: $t_i = \infty$ for each vertex $i \in V$
 3: $t_s = 0$ and $pred(s) = 0$
 4: while $|S| < n$ do
 5:    i = FindMinArrTimedUnvisitedNode_Cuda
 6:    $S \leftarrow S \cup \{i\}$
 7:    $U \leftarrow U - \{i\}$
 8:    for each$(i, j) \in A(i)$ do
 9:       if $t_j > ArrivalTime(t_i, (i, j))$ then
10:          $t_j = ArrivalTime(t_i, (i, j))$ and $pred(j) \leftarrow i$
11:       end if
12:    end for
13: end while
```

Figure 4.6. Modified Dijkstra Cuda implementation.

```
Function FindMinArrTimedUnvisitedNode_Cuda

 1: $iterator\ iSmallest,\ iBegin$
 2: $iBegin \leftarrow nodes.begin()$
 3: $iSmallest \leftarrow thrust :: min\_element(nodesD, Compare\_Key\_Value)$
 4: return $thrust :: distance(iBegin, iSmallest)$
```

Figure 4.7. Function for finding an unvisited node with minimum arrival time using Cuda/Thrust.

```
//binary predicate used by thrust
struct Compare_Key_Value
{
  bool operator()(const Node &lhs,const Node &rhs)const
  {
    if(rhs.isVisited)
      return true;
    if(lhs.isVisited)
      return false;
    return (lhs.minArrTime < rhs.minArrTime);
  }
}
```

Figure 4.8. Compare Key Value structure.

4.5. OpenMP and Cuda Hybrid Implementation

We also provide a hybrid implementation of shortest path algorithm based on both Cuda and OpenMP. In this version, we store the graph in nodes and nodesD vectors because one is used from OpenMP and the other one is used from Cuda functions. We use Cuda for finding the unvisited element with minimum distance. We use *thrust* :: *min_element* function which is defined in thrust/extrema class in a similar fashion as we use it in our Cuda implementation. On the other hand, this hybrid implementation differs from the Cuda version in that, it uses the returned result to get the position of the element in our *nodes* vector. Then, it uses OpenMP to update the arrival times of the neighbours of the current node. Finally, it updates the *nodesD* vector by only updating the changed members of the *nodes* vector. Note that, the update of the Cuda vector *nodesD* is not in the OpenMP parallel code region. Hence there are no race conditions among the threads while the Cuda vector is updated.

Hybrid algorithm runs faster than the Cuda version because it uses OpenMP parallelization in the process of updating neighbors of the minimum distanced unvisited node. This is a small improvement when we consider very large graphs, since most of the

Algorithm Cuda and OpenMP (Hybrid) Modified Dijsktra

```
1:  $S \leftarrow \emptyset; U \leftarrow V$
2:  $t_i = \infty$ for each vertex $i \in V$
3:  $t_s = 0$ and $pred(s) = 0$
4:  $thrust :: copy(nodesD, nodes)$
5:  while $|S| < n$ do
6:     i = FindMinArrTimedUnvisitedNode_Cuda
7:     $S \leftarrow S \cup \{i\}$
8:     $U \leftarrow U - \{i\}$
9:     #pragma omp parallel shared{$node_i$} private{$node_j$}
10:    for each$(i, j) \in A(i)$ do
11:       if $t_j > ArrivalTime(t_i, (i, j))$ then
12:          $t_j = ArrivalTime(t_i, (i, j))$ and $pred(j) \leftarrow i$
13:       end if
14:    end for

15:    for each$(i, j) \in A(i)$ do
16:       $nodesD.at(j.position) \leftarrow nodes.at(j.position)$
17:    end for
18: end while
```

Figure 4.9. Modified Dijkstra Hybrid implementation.

running time is spent on the finding minimum distanced element. As will be shown in the next experiments and results chapter, this algorithm runs faster than the OpenMP implementation too. However, the hybrid version has a drawback ; it needs to store the graph data both in *std* :: *vector* on the host as well as in *thrust* :: *device_vector* on the GPU. Therefore, it requires twice as much memory than the two versions.

5. EXPERIMENTS AND RESULTS

Our goal is to improve the running time performance of the MD algorithm given by Sung *et al.* in [4]. Performance is the main issue of TDSPP algorithms, so we try to reduce algorithm run time using parallel processing. We compare our parallel implementations with our implementation of the sequential MD algorithm. We give metrics that we use to evaluate performance our parallel algorithms. We provide information on the tests and the hardware environment on which we carried out our tests. After that, we show the results obtained from our tests with respect to the performance metrics.

5.1. Performance Metrics and the Test Environment

In this section, we describe performance metrics, and test environment and input parameters we used in our test. We illustrate the performance of our parallel implementations using the following metrics, which are defined in Section 2.3:

- Running time of the three implementations,
- Speedup of parallel implementations over the sequential implementation,
- Efficiency of our OpenMP implementation,
- Scalability of our OpenMP implementation.

Testing environment specifications are in the Table 5.1. In the result figures below, unless it is stated otherwise, OpenMP and hybrid implementations are tested using all of the 12 processors of the computer. We do not turn on optimization parameters while compiling and running our implementations.

5.2. Input Graph Generation

For our test cases, we use Graph500 reference code to generate random graphs. This code generates a graph with vertices and edges between them. We provide two

Table 5.1. Test environment specifications.

Parameter	Computer Specifications
Operating System	Debian 7.9 - i686 GNU/Linux
Host Memory	8 GB Memory
Host CPU	12 Cores - Intel(R) Xeon(R) CPU X5660 @ 2.80GHz
Accelerator	NVIDIA Corporation GF110 [GeForce GTX 580]

inputs to the graph generation algorithm:

(i) The number of vertices to be generated in base 2 logarithm : We create test graphs by giving this input in the range 10 to 22 (i.e. number of vertices in the range 2^{10} to 2^{22}).

(ii) Maximum number of edges to be generated: We provide this input as 4,8,16,32,64 times the number of vertices to be generated.

After creating graphs with the Graph500 program, we use the DFS algorithm, described in Section 2.1.2, to find reachable nodes from the first node. After that, we remove all the unreachable nodes from graph before running any algorithm. The information about the generated graphs using Graph500 and the resulting graphs after removing unreachable nodes can be found at Appendix A.

Then, we set random integer weights (edge lengths) for each edge in the range 1 to 10 units using the C *rand*() function. To be able to test time dependent graphs, we generate random integer velocities in the range 1 to 4 (units per time interval) on edges so that the velocity of a car going through an edge changes in every time interval. This interval can be set as well, but in our tests we set the time interval to 1.

5.3. Results

In this section we give the results of our tests for the metrics of run time, speedup, efficiency and scalability. In the tables about graph sizes used in our tests, $|V_R|$ means the number of reachable nodes and $|E_R|$ means the remaining number of edges after unreachable nodes are removed. Note that, all results of our tests can be found in Appendix B.

5.3.1. Run Times Observations for Parallel Algorithms

Firstly, we test our parallel algorithms under different sizes of graphs to observe the change in their running times. We achieve this by running our algorithms with increasing numbers of nodes and edges in the input graph. In order to observe the performance due to the change in the number of edges we used graphs which have the *same ratio* of the number of edges to the number of nodes in (in this case, the ratio is approximately 8). The graphs used in Figure 5.1 and Figure 5.2 are: G20, G24, G35, G37, G41, G42 and G45. More information about these graphs can be found in Appendix A.

In Figure 5.1, run times of the Cuda, OpenMP and hybrid algorithms are plotted against the different number of edges in the input graph. It is inferred that OpenMP algorithm runs faster than the other implementations involving Cuda in small and medium sized inputs where the number of edges changes from 200K to 4M. However, when the number of edges go up to 5M, the Cuda implementations run faster than the OpenMP version. As can be seen in the rest of the results, run times of the Cuda and hybrid algorithms are very close to each other. In the Figures 5.1 and 5.2, plots of these algorithms overlap.

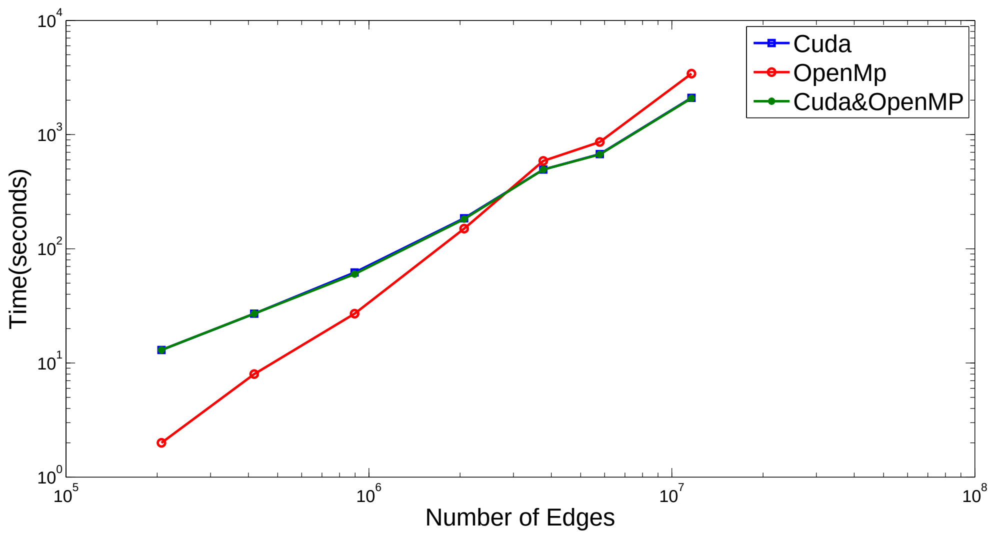


Figure 5.1. Parallel algorithms' run times with respect to the number of edges in graph.

In the Figure 5.2, it can be observed that the run times of algorithms increase when the number of nodes grows. It can be seen that OpenMP runs faster than Cuda and hybrid algorithm in small and medium sized graphs. However, after the number of vertices reaches 500K and more then the OpenMP implementation takes more time than the two Cuda based algorithms. In this figure, the ratio between the number of edges and the number of nodes is preserved as in Figure 5.1.

5.3.2. Comparison of Algorithms' Run Times

We test the sequential, OpenMP, Cuda and hybrid algorithms with respect to different size of graphs. Input graphs used in this section's figures are shown in Table 5.2 and Table 5.3 respectively, and also detailed information about the graphs and their run times can be found in Appendix A and B.

In Figure 5.3, we test those four algorithms with small and medium sized graphs. In the very small graphs the sequential algorithm runs faster than the Cuda and hybrid algorithm. However, when we increase the graph size a little, we can see that the

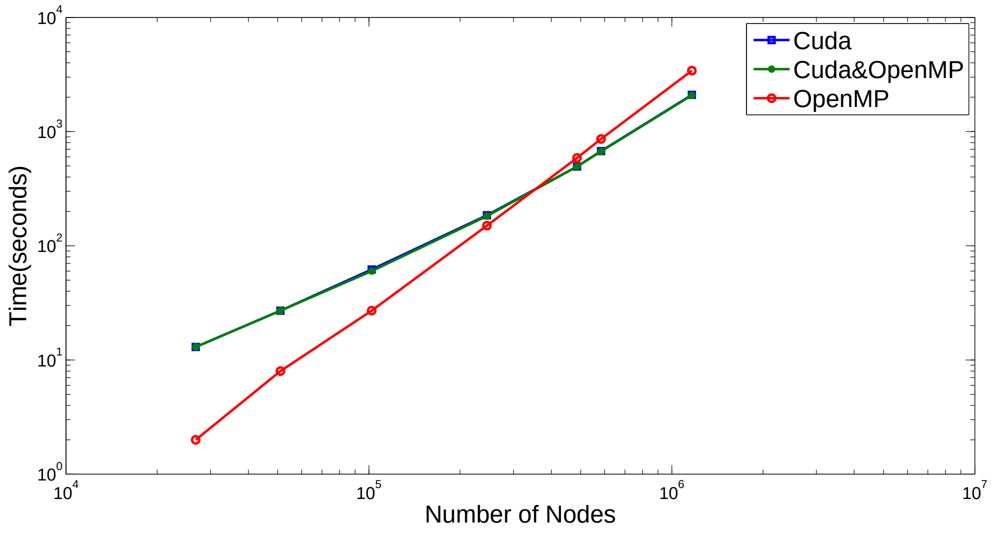


Figure 5.2. Parallel algorithms' run times with respect to the number of nodes in graph.

sequential algorithm run time is increasing rapidly with the graph size, but the parallel algorithms' run times does not increase as much as the sequential algorithm. It can also be seen that, although the run time of OpenMP algorithm is smaller then the rest of the algorithms, OpenMP algorithm's run time increases more than the Cuda algorithms.

In Figure 5.4, we try to observe the algorithms' run times on large graphs where the number of edges vary between 4M to 24M. As can be seen in Figure 5.4, OpenMP algorithm runs slightly slower than two other parallel algorithms. On the other hand, Cuda and hybrid(Cuda and OpenMP) algorithms' run times are almost the same. The difference between those algorithms is because of the neighbor update process after finding the minimum distanced unvisited node, which can be seen from Figure 4.9. If the number of neighbours to update is small for a node, the OpenMP parallel instructions cannot achieve a good efficiency due to small parallelism and thread management overhead.

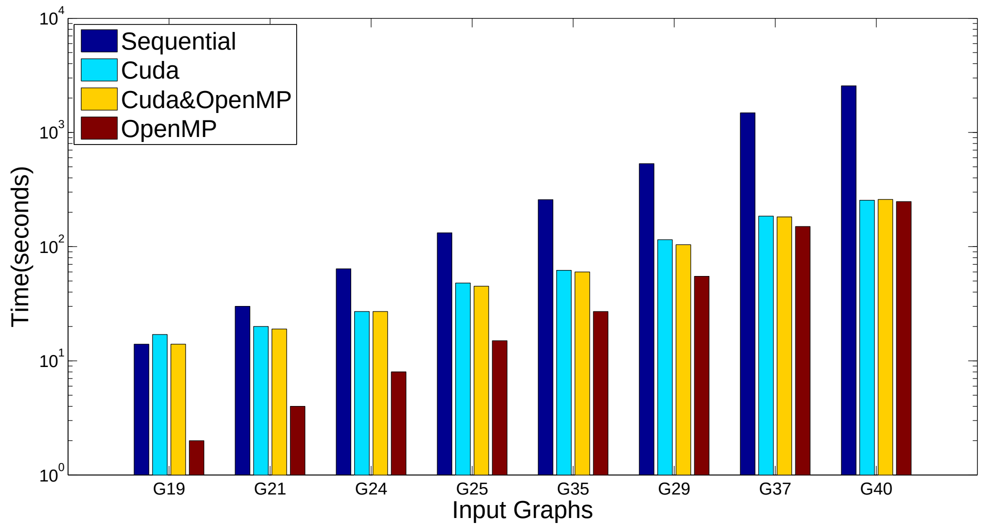


Figure 5.3. Algorithm run times for small and medium sized graphs.

Table 5.2. Small and medium sized graphs used in Figure 5.3.

Graph	$\|V_R\|$	$\|E_R\|$
G19	22,740	628,033
G21	35,069	450,770
G24	51,030	418,251
G25	72,737	1,124,437
G35	102,093	896,498
G29	146,411	2,636,679
G37	245,143	2,061,600
G40	319,769	1,567,005

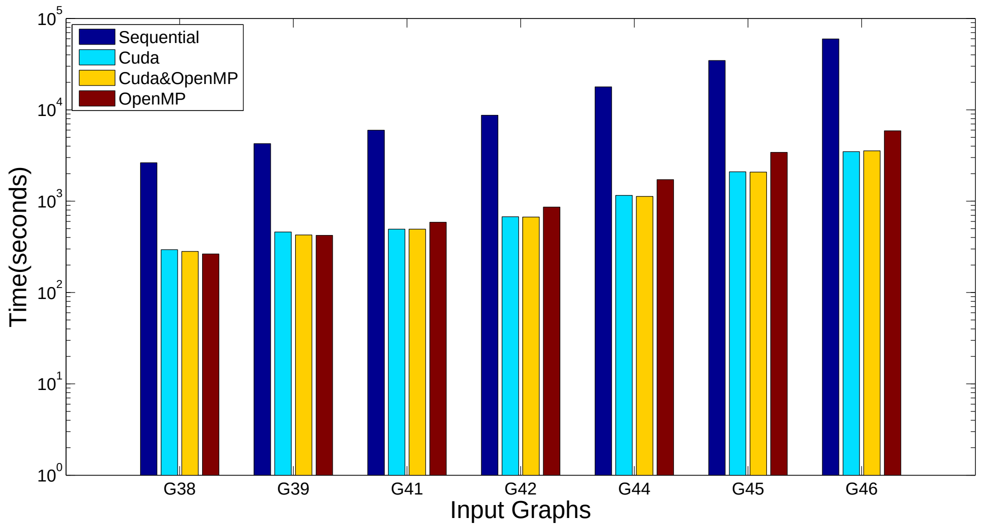


Figure 5.4. Algorithm run times for large graphs.

Table 5.3. Large graphs used in Figure 5.4.

Graph	$\|V_R\|$	$\|E_R\|$
G38	323,929	4,314,274
G39	409,860	9,007,004
G41	486,299	3,762,438
G42	583,850	5,785,179
G44	834,701	5,385,893
G45	1,164,208	11,606,886
G46	1,527,605	23,990,183

5.3.3. Speedup

When evaluating speedup performance, we consider all graphs used in our tests except for those which has a very small run time close to zero (i.e. ε). Therefore, we use graphs $G17 - G46$ given in Table A.1. First we order the graphs with respect to their run times in sequential algorithm. The graph with the smallest run time is the first element of the list. The ordered list of graphs with respect to the sequential algorithm run times for the set of $G17 - G46$ are: $G17$, $G18$, $G19$, $G20$, $G32$, $G21$, $G22$, $G23$, $G33$, $G24$, $G25$, $G34$, $G26$, $G27$, $G35$, $G28$, $G29$, $G36$, $G30$, $G31$, $G37$, $G40$, $G38$, $G39$, $G41$, $G42$, $G44$, $G43$, $G45$ and $G46$ respectively.

In Figure 5.5, the parallel algorithm speedups with respect to the increasing order of sequential run time are plotted. The ratio of the number of edges divided by the number of vertices for each graph is also presented in this figure. OpenMP algorithm starts with a speedup value of around 7 on small graphs. After the graph size increases, OpenMP algorithm reaches its maximum speedup which is just above 10. On the other hand, it can be seen from this figure that Cuda and hybrid algorithm starts with a speedup of 1 and ends up with a speedup of 17. As the graph size increases the ratio of run times between sequential and the Cuda algorithms increase. Also, note that speedups of the the hybrid algorithm and the Cuda algorithm are close to each other. This is because the difference between those two algorithms is the updating of the arrival times of the neighbours after finding the node with the smallest arrival time. Therefore, the difference between speedups of these two Cuda algorithms increases when there are more neighbors to update, which also means when the ratio of the number of edges to the number of vertices increases.

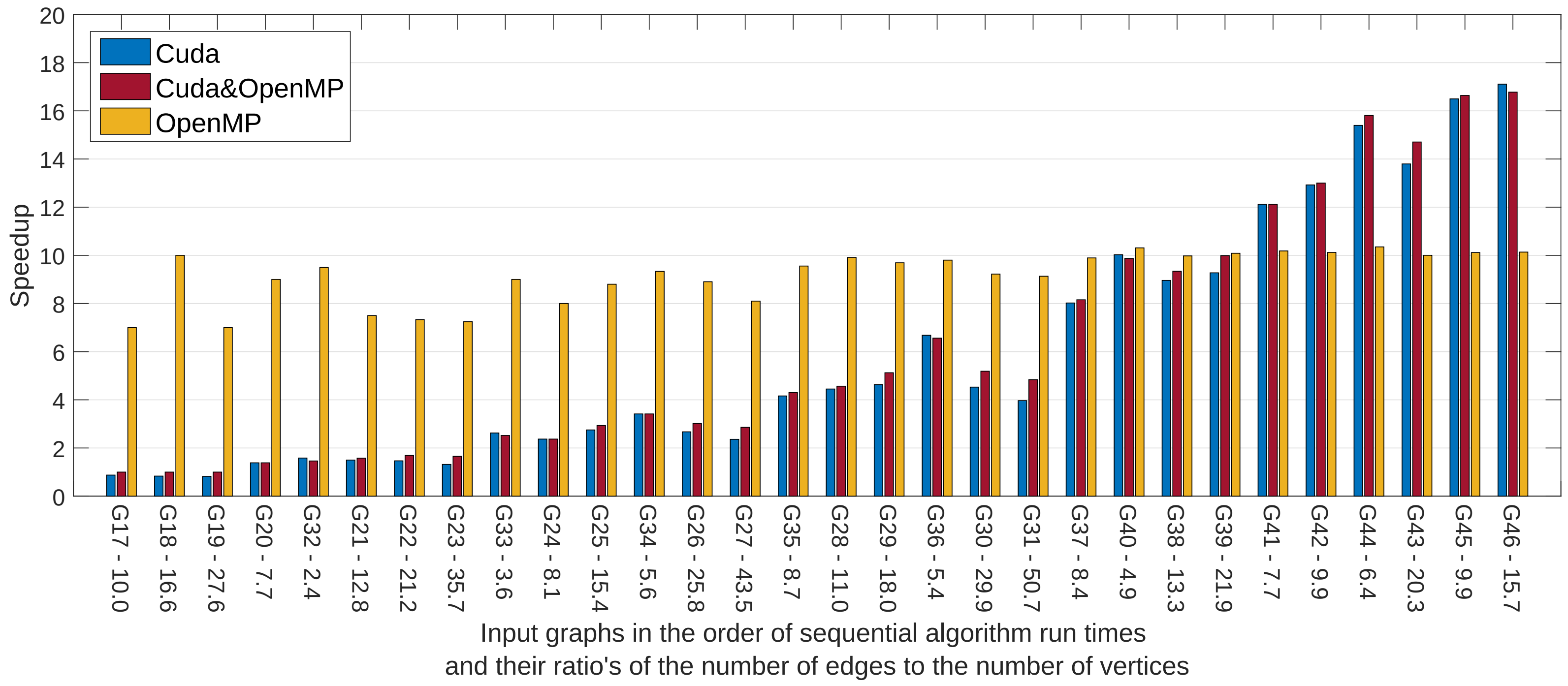
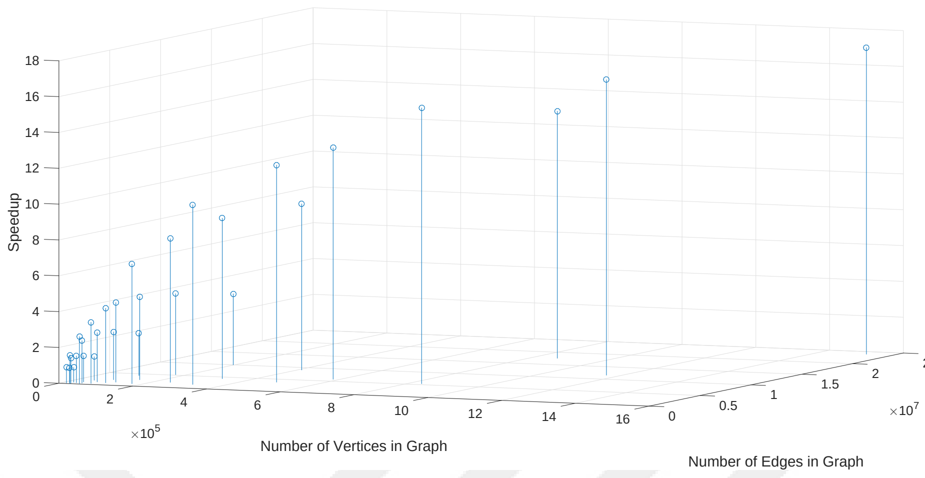


Figure 5.5. Speedup of parallel algorithms against sequential algorithm.

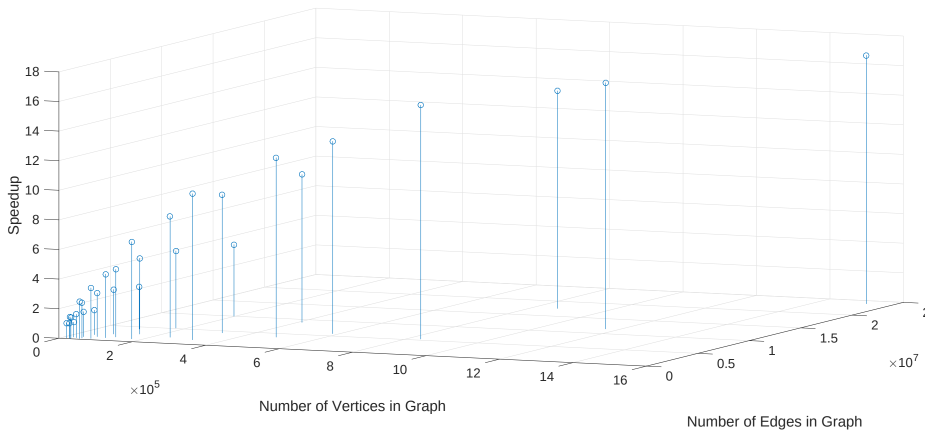
In Figure 5.6, speedups of our three parallel implementations are shown with respect to the number of vertices and the number of edges in graphs using the same graphs in Figure 5.5. In Figures 5.5 and 5.6, fluctuations in speedups of Cuda and hybrid results are because of the ratio of number of edges to the number of vertices in the graph changes. An increase in this ratio causes the speedups of Cuda and hybrid implementations to decrease as the graph size increased, because the Cuda implementations mostly benefit from finding the minimum distanced unvisited node. So, an increase in number of vertices while the number of edges does not change has more speedup effect on Cuda algorithms than an increase in the number of edges while the number of vertices does not change in a graph. Note that, hybrid implementation uses OpenMP when updating neighbours of the minimum distanced node, therefore, the hybrid implementation's speedup does not decrease as much as the the Cuda implementation speedup while the ratio of number of edges to the number of vertices increasing. However, in the hybrid implementation there is a copy overhead from host vector to device vector after the update process, which also results in decrease in the speedup. Note that, OpenMP implementation speedup is also decrease when the ratio of the number of edges to the number of nodes increases, but, because it has no copy overhead as in the hybrid implementation, the effect is smaller than those in other parallel algorithms.

5.3.3.1. Relative Speedup. We observe the OpenMP algorithm speedup results with different number of processors to work with. We test the small, the medium and the big sized graphs using 1, 2, 4, 6, 8, 10 and 12 number of threads. The results are shown in Figure 5.7. We order the graphs in between $G17 - G46$ in ascending order of the sequential algorithm run time as we did in Section 5.3.3. The graph with the smallest run time is the first element of the list. We create 3 sets using this list:

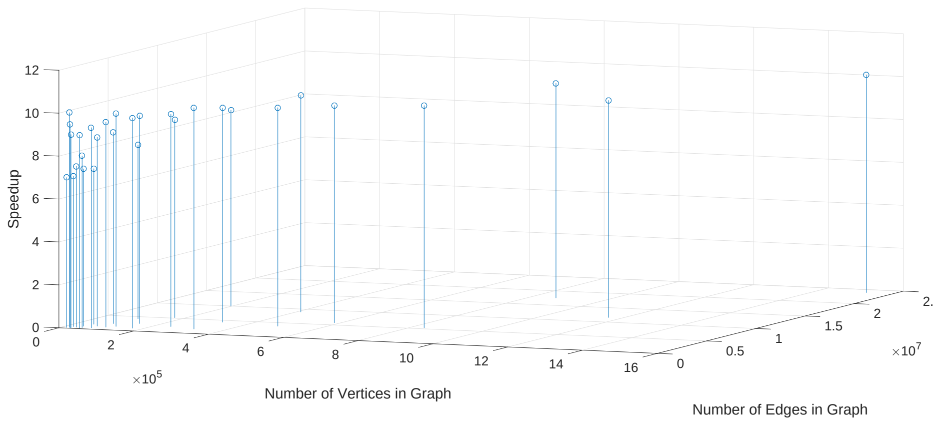
(i) Small Graphs: $G17$, $G18$, $G19$, $G20$, $G32$, $G21$, $G22$, $G23$, $G33$ and $G24$
(ii) Medium Graphs: $G25$, $G34$, $G26$, $G27$, $G35$, $G28$, $G29$, $G36$, $G30$ and $G31$
(iii) Large Graphs: $G37$, $G40$, $G38$, $G39$, $G41$, $G42$, $G44$, $G43$, $G45$ and $G46$,



(a) Cuda



(b) Cuda&OpenMP



(c) OpenMP

Figure 5.6. Speedups of parallel algorithms with respect to the number of vertices and the number of edges in graphs.

After that, we calculate the average running time of the graphs in each set for the OpenMP algorithm with using 1,2,4,6,8,10,12 number of threads respectively. Then, we calculate OpenMP algorithm speedups for each graph set with the thread counts of 1,2,4,6,8,10,12. We reach up to 10 times faster running time on 12 threads than what we have with 1 thread. It can be seen in Figure 5.7 that, in particular for medium and large graphs, the behaviour of OpenMP algorithm run time is similar to the change in the number of threads. As can be seen from this figure, the number of threads and OpenMP algorithm speedup are roughly linearly proportional to each other. Therefore, it can be expected that increasing the number of processors even after 12 will continue to achieve a good efficiency in terms of runtime.

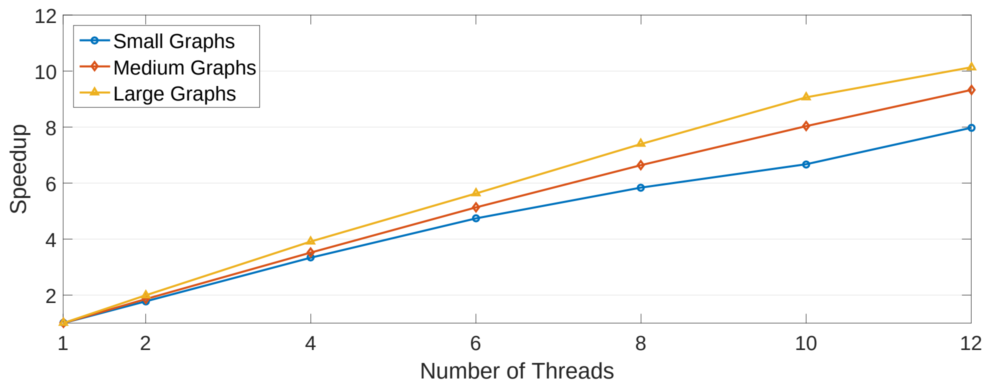


Figure 5.7. OpenMP Algorithm speedups with respect to the number of threads used.

5.3.4. Efficiency

In our OpenMP implementations we also examine the efficiency metric which is defined as the ratio of speedup to the number of processors used in program run time. In an ideal system, the efficiency should be one.

In the Figure 5.8, the following notation is used for legends:

- $OpenMP_{12}$: OpenMP algorithm with 12 threads used.
- $OpenMP_{10}$: OpenMP algorithm with 10 threads used.
- $OpenMP_8$: OpenMP algorithm with 8 threads used.

- $OpenMP_6$: OpenMP algorithm with 6 threads used.
- $OpenMP_4$: OpenMP algorithm with 4 threads used.
- $OpenMP_2$: OpenMP algorithm with 2 threads used.

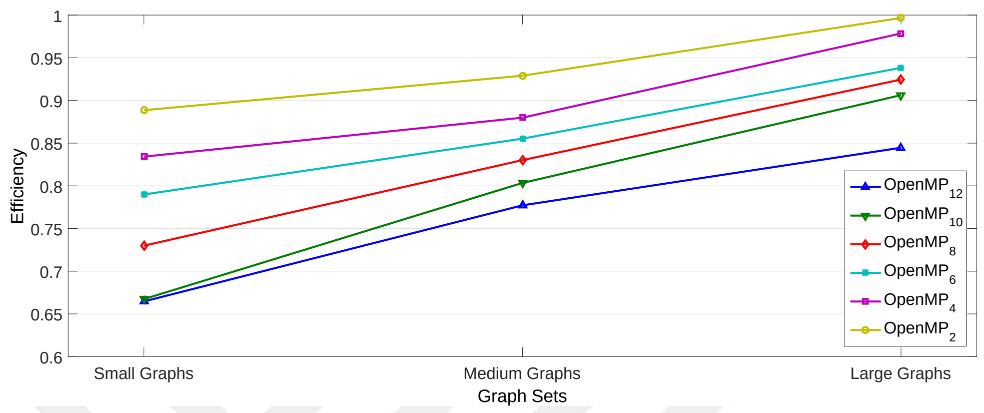


Figure 5.8. OpenMP Algorithm efficiency with respect to the number of threads used.

We use the same graph input sets (Small, Medium and Large Graphs) that we use in Section 5.3.3.1). The efficiency of our OpenMP implementation is shown in Figure 5.8. Every line in the graph corresponds to an OpenMP algorithm efficiency with a constant number of threads. The following results can be inferred from this figure:

(i) For Large Graphs set, all of the efficiency results for different number of threads that we have tested are above or close 0.85, which is a good efficiency result.

(ii) For Medium Graphs set, all of the efficiency results for different number of threads that we have tested are above or close 0.75.

(iii) For Small Graphs set, all of the efficiency results for different number of threads that we have tested are above or close 0.65.

(iv) For each line, the results for each thread count, as the input size grows, the efficiency increases too. This means that as the graph size increase the total idle time for threads decrease.

5.3.5. Scalability

Scalability measures the performance of the program while the input and the processor size increases at the same ratio. Because that the graphs we are testing are the graphs with unreachable nodes removed, it is hard to produce those graphs which have the same ratio of number of nodes to number of edges as in Graph500.

Therefore, we use 3 graphs, namely $G20$, $G24$ and $G35$, to examine our OpenMP implementation's scalability. Those graphs have twice as many number of reachable nodes and edges in the given order above. Then, we compare the efficiency of these graphs under 2, 4 and 8 threads respectively. Therefore, as we increase the graph size twice, we increase the number of threads twice as many too.

The results are shown in the Table 5.4. In this table, T_s is the sequential algorithm run time in seconds, T_p is the OpenMP algorithm run time in seconds with given number of threads and η is the efficiency.

Table 5.4. Scalability of OpenMP Algorithm.

Graph	$\|V_R\|$	$\|E_R\|$	$\|E_R\|/\|V_R\|$	# of threads	T_s	T_p	η
G20	26,798	206,590	7.7	2	18	11	0.81
G24	51,030	418,251	8.1	4	64	19	0.84
G35	102,093	896,498	8.7	8	258	39	0.82

Results show that, as we increase the graph size and the number of processor by 2 and efficiency of algorithm is almost the same (range in between 0,81 and 0,84).

5.4. Discussion

OpenMP algorithm runs faster than sequential algorithm without taking into consideration of graph size. However, there is an increase in the algorithm speedup from 7 up to 10 while we increase the input graph size. This is because increasing graph size makes the program run more efficiently so that the total idle time of the processors with respect to the total run time will decrease.

Cuda and hybrid algorithms mostly benefit from finding the unvisited node with minimum arrival time faster than the OpenMP algorithm. As we can see in the figures at Section 5.3, Cuda algorithms increase the speedup ratios when graph size increases up to 17. On the other hand, as the graph size increases the speedup gap between Cuda and Cuda&OpenMP algorithms also increases. This is because, there is an additional parallelization section in the hybrid algorithm so that arrival times of the neighbors of the unvisited minimum arrival timed nodes are also updated in parallel. When the number of neighbors is small, the effect of parallelization can not be perceived because there is also an overhead of the copying all of the updated nodes to the $thrust :: device\_vector$.

6. CONCLUSIONS

In this thesis, we present efficient parallel algorithms for shortest path problem from one node to all other nodes on time dependent graphs. We implement parallel algorithms in Cuda, OpenMP and hybrid Cuda and OpenMP. We evaluate these algorithms in terms of algorithm run times with respect to the sequential algorithm. For testing, we use an extensive set of graphs which have number of edges between 200K and 24M, and have number of nodes between 22K and 1.5M. We achieve up to 10-fold speedup with 12 processor in our OpenMP implementation, and 17-fold speedup in Cuda implementations. It is observed that OpenMP implementation has better performance when the number of nodes in the graph is around or smaller than 400K. On the other hand, Cuda and hybrid algorithms are good choices when the number of nodes in the graph is larger than 450K. Also, we can say that as the ratio between number of edges and number of nodes in a graph increases the hybrid algorithm becomes a little more efficient than the Cuda algorithm. If the memory constraint is not an issue, then hybrid algorithm can be a preferred solution on dense graphs.

In this thesis, we are mainly concerned about the parallelization of the shortest path problem with time dependent flow speed model on TDG. Our efforts focused on parallelization and not on optimization of the original sequential algorithm. In sequential implementations, heap based data structures provide faster mechanism for finding minimum elements in traditional shortest path problems on non-time dependent graphs. Due to:

(i) Requirement of the MD algorithm to update nodes' minimum arrival times (shown between 8-12 lines on Figure 3.2): If a heap structure is used to store minimum arrival times for nodes, then, updating a node's minimum arrival time would require that updated node is searched in the heap and removed, and after that new updated value inserted to the heap. Since, searching operation is very costly in heap structures,

(ii) The fact that heap data structures introduce synchronization overheads when parallel accesses are performed,

we did not use heaps in our parallel algorithms. When testing performance, we use as sequential algorithm, our parallel implementation running on one thread. As a future work, we aim to optimize the sequential MD algorithm implementation on time dependent flow speed model using self-balancing search trees. We will store the node id's with the key of their current minimum distances in the self-balancing search trees. This will improve the search time to find the unvisited node with minimum distance. We also intend to evaluate this approach in our parallel implementations.

APPENDIX A: TEST GRAPHS INFORMATION

In this Appendix, we give our generated graphs' information. For the legends in table headers, following notation is used:

- Graph : Input graph to be used in tests.
- log_V : log_2 of number of vertices to be generated.
- $|E|/|V|$: The ratio of number of edges divided by number of vertices to be created.
- $|V_C|$: Number of vertices created.
- $|E_C|$: Number of edges created.
- $|V_R|$: Number of reachable vertices.
- $|E_R|$: Number of reachable edges.
- $|E_R|/|V_R|$: The ratio of the number of reachable edges divided by the number of reachable vertices(one digit shown after the decimal separator).

We used graph500 [15] to generate our test graphs. Input of the reference code is log of number of vertices to be generated and the limit of edges to be created. So, $log_{V_{toBeGenerated}}$ and $|E|/|V|$ are used in graph500 program. $|V_C|$ and $|E_C|$ informations are about the generated graph. $|V_R|$ and $|E_R|$ are the remaining number of vertices and edges after we remove the unreachable nodes in graphs.

Table A.1. Test graphs information.

Graph	log_V	$\|E\|/\|V\|$	$\|V_C\|$	$\|E_C\|$	$\|V_R\|$	$\|E_R\|$	$\|E_R\|/\|V_R\|$
G01	10	8	805	6,072	112	168	1.5
G02	10	16	880	10,624	198	428	2.1
G03	10	32	946	17,846	326	1,327	4.0
G04	10	64	986	28,798	503	4,038	8.0
G05	11	8	1,540	12,810	556	1,587	2.8
G06	11	16	1,716	22,882	869	4,655	5.3
G07	11	32	1,855	39,278	1,091	9,064	8.3
G08	11	64	1,952	65,365	1,380	20,670	14.9
G09	13	16	6,466	102,301	892	2,095	2.3
G10	13	32	7,099	183,486	2,300	12,773	5.5
G11	13	64	7,529	319,739	3,138	30,235	9.6
G12	14	8	10,965	114,243	4,199	17,731	4.2
G13	14	16	12,603	213,074	5,979	39,127	6.5
G14	14	32	13,816	388,548	7,541	78,399	10.4
G15	14	64	14,826	690,810	8,943	152,948	17.1
G16	15	8	21,079	233,974	12,256	76,896	6.2

Table A.1. Test graphs information (cont.).

Graph	log_V	$\|E\|/\|V\|$	$\|V_C\|$	$\|E_C\|$	$\|V_R\|$	$\|E_R\|$	$\|E_R\|/\|V_R\|$
G17	15	16	24,203	441,320	16,036	161,832	10.0
G18	15	32	26,942	816,631	19,654	327,037	16.6
G19	15	64	29,017	1,474,812	22,740	628,033	27.6
G20	16	8	40,398	477,504	26,798	206,590	7.7
G21	16	16	46,836	909,685	35,069	450,770	12.8
G22	16	32	52,292	1,701,797	42,087	895,957	21.2
G23	16	64	56,796	3,118,334	48,149	1,722,340	35.7
G24	17	8	77,522	971,405	51,030	418,251	8.1
G25	17	16	90,332	1,864,629	72,737	1,124,437	15.4
G26	17	32	101,534	3,523,793	86,495	2,236,552	25.8
G27	17	64	110,874	6,535,667	98,567	4,292,432	43.5
G28	18	8	148,833	1,969,774	118,329	1,307,774	11.0
G29	18	16	174,017	3,806,052	146,411	2,636,679	18.0
G30	18	32	197,033	7,253,209	173,399	5,188,036	29.9
G31	18	64	216,447	13,601,052	197,090	9,992,620	50.7
G32	19	8	285,234	3,985,147	28,023	67,795	2.4

Table A.1. Test graphs information (cont.).

Graph	log_V	$\|E\|/\|V\|$	$\|V_C\|$	$\|E_C\|$	$\|V_R\|$	$\|E_R\|$	$\|E_R\|/\|V_R\|$
G33	19	16	335,774	7,741,316	50,467	186,147	3.6
G34	19	32	381,828	14,862,744	75,304	424,621	5.6
G35	19	64	421,772	28,136,604	102,093	896,498	8.7
G36	20	8	547,314	8,042,773	172,325	933,288	5.4
G37	20	16	646,508	15,699,276	245,143	2,061,600	8.4
G38	20	32	739,428	30,341,529	323,929	4,314,274	13.3
G39	20	64	821,615	57,898,073	409,860	9,007,004	21.9
G40	21	4	853,956	8,214,909	319,769	1,567,005	4.9
G41	21	8	1,049,061	16,207,313	486,299	3,762,438	7.7
G42	21	16	1,244,280	31,767,836	583,850	5,785,179	9.9
G43	21	32	1,431,016	61,722,713	866,253	17,585,481	20.3
G44	22	4	1,630,278	16,492,891	834,701	5,385,893	6.4
G45	22	8	2,009,000	32,620,535	1,164,208	11,606,886	9.9
G46	22	16	2,396,019	64,153,052	1,527,605	23,990,183	15.7

APPENDIX B: TEST RESULTS

In this appendix, we give the explicit results of all tests we have run in order to observe the performance of our implementations.

For the legends in table headers, following notation is used:

- Graph : Input graph used in tests.
- Sequential : Sequential algorithm run time in seconds.
- Cuda : Cuda algorithm run time in seconds.
- Cuda&OpenMP12 : Cuda&OpenMP algorithm run time in seconds in which 12 processors used.
- OpenMP12 : OpenMP algorithm run time in seconds in which 12 processors used.
- OpenMP10 : OpenMP algorithm run time in seconds in which 10 processors used.
- OpenMP8 : OpenMP algorithm run time in seconds in which 8 processors used.
- OpenMP6 : OpenMP algorithm run time in seconds in which 6 processors used.
- OpenMP4 : OpenMP algorithm run time in seconds in which 4 processors used.
- OpenMP2 : OpenMP algorithm run time in seconds in which 2 processors used.

Information about graphs used in our tests can be found at Table A.1. Note that, ε is used for the test results in the range of $0 < \varepsilon < 1$.

Table B.1. Sequential and parallel algorithms run times in seconds.

Graph	Sequential	Cuda	Cuda&OpenMP12	OpenMP12	OpenMP10	OpenMP8	OpenMP6	OpenMP4	OpenMP2
G01	ε	ε	1	ε	ε	ε	ε	ε	ε
G02	ε	ε	ε	ε	ε	ε	ε	ε	ε
G03	ε	ε	ε	ε	ε	ε	ε	ε	ε
G04	ε	1	ε	ε	ε	ε	ε	ε	ε
G05	ε	ε	ε	ε	ε	ε	ε	ε	ε
G06	ε	ε	ε	ε	ε	ε	ε	ε	1
G07	ε	1	1	ε	ε	ε	ε	ε	ε
G08	1	ε	ε	ε	1	ε	ε	ε	ε
G09	ε	ε	ε	ε	ε	ε	ε	ε	ε
G10	ε	1	1	1	ε	ε	ε	ε	ε
G11	ε	2	1	ε	ε	ε	1	ε	ε
G12	1	1	2	ε	ε	ε	ε	ε	ε
G13	1	3	2	ε	ε	1	ε	ε	ε
G14	1	3	3	1	ε	ε	1	1	1
G15	2	5	4	1	1	ε	1	1	1
G16	4	5	5	ε	1	1	1	1	2

Table B.1. Sequential and parallel algorithms run times in seconds (cont.).

Graph	Sequential	Cuda	Cuda&OpenMP12	OpenMP12	OpenMP10	OpenMP8	OpenMP6	OpenMP4	OpenMP2
G17	7	8	7	1	1	2	2	2	4
G18	10	12	10	1	2	2	2	3	6
G19	14	17	14	2	3	3	3	4	8
G20	18	13	13	2	3	3	4	5	11
G21	30	20	19	4	4	6	7	10	17
G22	44	30	26	6	7	8	10	14	26
G23	58	44	35	8	10	10	14	19	34
G24	64	27	27	8	8	10	12	19	36
G25	132	48	45	15	17	20	27	38	73
G26	187	70	62	21	25	30	38	55	107
G27	243	103	85	30	34	39	51	74	140
G28	347	78	76	35	43	52	65	97	190
G29	533	115	104	55	66	79	102	152	289
G30	747	165	144	81	92	114	146	215	413
G31	968	244	200	106	123	148	193	279	528

Table B.1. Sequential and parallel algorithms run times in seconds (cont.).

Graph	Sequential	Cuda	Cuda&OpenMP12	OpenMP12	OpenMP10	OpenMP8	OpenMP6	OpenMP4	OpenMP2
G32	19	12	13	2	3	3	4	5	10
G33	63	24	25	7	8	9	11	17	32
G34	140	41	41	15	18	21	26	40	72
G35	258	62	60	27	32	39	51	72	133
G36	735	110	112	75	84	104	137	197	364
G37	1,484	185	182	150	179	216	279	390	743
G38	2,634	294	282	264	304	373	479	692	1,347
G39	4,266	460	427	423	500	586	782	1,127	2,141
G40	2,557	255	259	248	288	360	466	646	1,240
G41	5,988	494	494	588	695	849	1,074	1,498	2,842
G42	8,724	675	671	862	951	1,152	1,534	2,181	4,153
G43	19,162	1,389	1,303	1,916	2,136	2,621	3,454	4,953	9,711
G44	17,814	1,157	1,127	1,721	1,928	2,386	3,120	4,426	8,674
G45	34,610	2,098	2,080	3,421	3,816	4,639	6,097	8,662	17,334
G46	59,673	3,488	3,557	5,888	6,518	8,034	10,589	15,516	30,535

REFERENCES

1. Larsen, S. ø., H. Koren and R. Solberg, "Traffic Monitoring Using Very High Resolution Satellite Imagery", *Photogrammetric Engineering & Remote Sensing*, Vol. 75, No. 7, pp. 859–869, 2009.

2. Ding, B., J. X. Yu and L. Qin, "Finding Time-Dependent Shortest Paths Over Large Graphs", *Proceedings of the 11Th International Conference on Extending Database Technology: Advances in Database Technology*, pp. 205–216, Acm, 2008.

3. Brodal, G. S. and R. Jacob, "Time-Dependent Networks as Models to Achieve Fast Exact Time-Table Queries", *Electronic Notes in Theoretical Computer Science*, Vol. 92, pp. 3–15, 2004.

4. Sung, K., M. G. Bell, M. Seong and S. Park, "Shortest Paths in a Network with Time-Dependent Flow Speeds", *European Journal of Operational Research*, Vol. 121, No. 1, pp. 32–39, 2000.

5. Dean, B. C., "Shortest Paths in Fifo Time-Dependent Networks: Theory and Algorithms", *Rapport Technique, Massachusetts Institute of Technology*, 2004.

6. Sherali, H. D., K. Ozbay and S. Subramanian, "The Time-Dependent Shortest Pair of Disjoint Paths Problem: Complexity, Models, and Algorithms", *Networks*, Vol. 31, No. 4, pp. 259–272, 1998.

7. Hauwert, G., *Time Dependent Optimization Problems in Networks*, Ph.D. Thesis, Universiteit Leiden, 2010.

8. Ohshima, T., *A Landmark Algorithm for the Time-Dependent Shortest Path Problem*, Ph.D. Thesis, Citeseer, 2008.

9. Goldberg, A. V. and C. Harrelson, "Computing the Shortest Path: A Search Meets

Graph Theory", *Proceedings of the Sixteenth Annual Acm-Siam Symposium on Discrete Algorithms*, pp. 156–165, Society for Industrial and Applied Mathematics, 2005.

10. Dreyfus, S. E., "An Appraisal of Some Shortest-Path Algorithms", *Operations Research*, Vol. 17, No. 3, pp. 395–412, 1969.

11. Kumar, V. and V. Singh, "Scalability of Parallel Algorithms for the All-Pairs Shortest-Path Problem", *Journal of Parallel and Distributed Computing*, Vol. 13, No. 2, pp. 124–138, 1991.

12. Chabini, I. and S. Ganugapati, "Parallel Algorithms for Dynamic Shortest Path Problems", *International Transactions in Operational Research*, Vol. 9, No. 3, pp. 279–302, 2002.

13. Tremblay, N. and M. Florian, "Temporal Shortest Paths: Parallel Computing Implementations", *Parallel Computing*, Vol. 27, No. 12, pp. 1569–1609, 2001.

14. Ziliaskopoulos, A., D. Kotzinos and H. S. Mahmassani, "Design and Implementation of Parallel Time-Dependent Least Time Path Algorithms for Intelligent Transportation Systems Applications", *Transportation Research Part C: Emerging Technologies*, Vol. 5, No. 2, pp. 95–107, 1997.

15. Bader, D., J. Berry, S. Kahan, R. Murphy, J. Riedy and J. Willcock, *The Graph 500 List: Graph 500 Reference Implementations*, 2010, `http://www.graph500.org/referencecode`, [Accessed November 2015].

16. Cormen, T. H., C. E. Leiserson, R. L. Rivest and C. Stein, "Introduction to Algorithms Second Edition", *The Knuth-Morris-Pratt Algorithm", Year*, 2001.

17. Kanoulas, E., Y. Du, T. Xia and D. Zhang, "Finding Fastest Paths on a Road Network with Speed Patterns", *Data Engineering, 2006. Icde'06. Proceedings of the 22Nd International Conference On*, pp. 10–10, IEEE, 2006.

18. Orda, A. and R. Rom, "Shortest-Path and Minimum-Delay Algorithms in Networks with Time-Dependent Edge-Length", *Journal of the Acm (Jacm)*, Vol. 37, No. 3, pp. 607–625, 1990.

19. Kaufman, D. E. and R. L. Smith, "Fastest Paths in Time-Dependent Networks for Intelligent Vehicle-Highway Systems Application∗", *Journal of Intelligent Transportation Systems*, Vol. 1, No. 1, pp. 1–11, 1993.

20. Dijkstra, E. W., "A Note on Two Problems in Connexion with Graphs", *Numerische Mathematik*, Vol. 1, No. 1, pp. 269–271, 1959.

21. Bang-Jensen, J. and G. Z. Gutin, *Digraphs: Theory, Algorithms and Applications*, Springer Science & Business Media, 2008.

22. Bellman, R., *On a Routing Problem*, Tech. rep., Dtic Document, 1956.

23. Ford Jr, L. R., *Network Flow Theory*, Tech. rep., Dtic Document, 1956.

24. Moore, E. F., *The Shortest Path Through a Maze*, Bell Telephone System., 1959.

25. Chabini, I., *A New Algorithm for Shortest Paths in Discrete Dynamic Networks*, Tech. rep., 1997.

26. Chabini, I., "Discrete Dynamic Shortest Path Problems in Transportation Applications: Complexity and Algorithms with Optimal Run Time", *Transportation Research Record: Journal of the Transportation Research Board*, , No. 1645, pp. 170–175, 1998.

27. Ganugpati, S. V., *Dynamic Shortest Paths Algorithms: Parallel Implementations and Application to the Solution of Dynamic Traffic Assignment Models*, Ph.D. Thesis, Massachusetts Institute of Technology, 1998.

28. Floyd, R. W., "Algorithm 97: Shortest Path", *Communications of the Acm*, Vol. 5, No. 6, p. 345, 1962.

29. Pettie, S., “A New Approach to All-Pairs Shortest Paths on Real-Weighted Graphs”, *Theoretical Computer Science*, Vol. 312, No. 1, pp. 47–74, 2004.

30. Sahni, S. and V. Thanvantri, *Parallel Computing: Performance Metrics and Models*, Tech. rep., 1995.

31. Grama, A., A. Gupta, G. Karypis and V. Kumar, “Introduction to Parallel Computing”, *Introduction to Parallel Computing, 2Nd Edn, by A. Grama Et Al. Pearson Education Limited, Harlow, England (Isbn: 978-0-201-64865-2)*, Vol. 1, 2003.

32. Gupta, A. and V. Kumar, “Isoefficiency Function: A Scalability Metric for Parallel Algorithms and Architectures”, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 8, pp. 922–932, 1993.

33. Ziliaskopoulos, A. K. and H. S. Mahmassani, “Time-Dependent, Shortest-Path Algorithm for Real-Time Intelligent Vehicle Highway System Applications”, *Transportation Research Record*, pp. 94–94, 1993.

34. Mitton, R., *Very Sleepy*, 2014, `http://www.codersnotes.com/sleepy/`, [Accessed November 2015].