GPU Algorithms for Fastest Path Problem in Temporal Graphs

Mithinti Srikanth
cs18d501@iittp.ac.in
Indian Institute of Technology Tirupati
Tirupati, Andhra Pradesh, India

Prashant Singh
cs22m110@iittp.ac.in
Indian Institute of Technology Tirupati
Tirupati, Andhra Pradesh, India

G. Ramakrishna
rama@iittp.ac.in
Indian Institute of Technology Tirupati
Tirupati, Andhra Pradesh, India

ABSTRACT

This paper introduces the first GPU-based parallel algorithms to solve the fastest path duration (FPD) problem in temporal graphs. Fastest path duration in temporal graphs is a well studied problem that has multiple use cases in information diffusion, epidemic spreading, and route planning in public transportation. Given a temporal graph G in which each edge associates with a departure time and duration time, and a source vertex s, the Fastest Path Duration (FPD) problem is to compute the journey times from s to all the rest of the vertices in G. The existing multi-core algorithm for FPD by Delling et al. exhibits limited parallelism. In general, many parallel algorithms suffer from doing redundant work while pruning certain computations. Our research focuses on multiple algorithmic ways to avoid redundant work and perform pruning of computations effectively. We introduce three novel GPU-based parallel algorithms for FPD, namely Level Order (LO), Multiple Breadth First Search (MBFS), and Local Work-lists (LW) and implement them on a GPU architecture machine. Our algorithms demonstrate an average speedup of approximately 165 times and a maximum speedup of up to 1383 times over the current state-of-the-art algorithms. This paper provides a comprehensive explanation of various algorithm designs, their optimizations for GPUs, and an extensive evaluation of their performance across various temporal graph scenarios.

CCS CONCEPTS

• **Computing methodologies** → **Parallel algorithms**; **Massively parallel algorithms**.

KEYWORDS

Temporal Graph, Fastest Path, GPU-Algorithms, Massively Parallel Algorithms, Transformed Graph, Journey Time

ACM Reference Format:
Mithinti Srikanth, Prashant Singh, and G. Ramakrishna. 2024. GPU Algorithms for Fastest Path Problem in Temporal Graphs. In *The 53rd International Conference on Parallel Processing (ICPP '24), August 12–15, 2024, Gotland, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3673038.3673105

ICPP '24, August 12–15, 2024, Gotland, Sweden

ACM ISBN 979-8-4007-1793-2/24/08
https://doi.org/10.1145/3673038.3673105

1 INTRODUCTION

In the contemporary world, graphs play an ubiquitous role in data structures. While traditional graphs are static, real-world scenarios often require temporal graphs to capture time-related changes. The limitations of static graphs are evident in today's world, especially with the rise of the internet and social networks, where temporal information is crucial. Thus, temporal graphs have become important in diverse and rapidly changing environments. In particular, path problems in temporal graphs are essential in traffic navigation [3], for optimizing routes based on real-time conditions and in social network analysis [2, 20] for tracking the rapid spread of information. Airlines use them to adjust schedules according to fluctuating traffic and weather [10], while supply chains rely on them for efficient routing amid variable conditions [13]. Path finding algorithms in temporal graphs are also crucial in emergency response for adaptable evacuation planning, urban planning for responsive transport systems [22], and telecommunications for maintaining consistent data flow [1]. They are invaluable in health informatics for disease modeling [16] and financial networks for optimizing transactions and detecting anomalies [14].

Graph analytics in temporal graphs are crucial for identifying key influencers, such as celebrities or notable persons, within a network. To do this effectively, centrality measures like betweenness and closeness are used. These measures are essential for determining the importance or influence of nodes in a graph. In particular, in a temporal graph, the fastest path duration (journey time) from a vertex to every other vertex determines its farness centrality and closeness centrality [12]. This requirement is essentially captured in the fastest path problem and is well-studied in the serial setting. However, with the increasing size of temporal graphs coming from real-world scenarios, there is a growing need for efficient parallel algorithms. These algorithms are necessary to process and analyze vast datasets by exploiting massive parallelism. Their development and implementation are crucial for managing the complexity and scale of modern graph analytics tasks.

A temporal graph $G = (V, E, T)$ is a directed weighted graph, where V is a set of vertices, E is a set of edges, and T associates time stamps on all the edges, indicating its availability. Each edge in a temporal graph can be represented by a tuple (u, v, t, λ), indicating an edge from u to v that starts at time t and λ specifies the duration required to travel from u to v, which is a positive integer. Here, t and $t + \lambda$ are referred to as departure time at u and arrival time at v, respectively.

A path formed by a sequence of edges is referred to as a *time respecting path* if the departure time at every intermediate vertex is at least the arrival time of the previous edge. Let P be a time-respecting path from a vertex s to vertex z, which adheres to the timing constraints. Let $dep(s)$ and $arr(z)$ denote the departure time

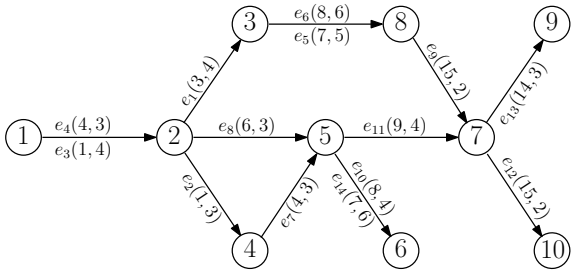

Figure 1: A simple Temporal Graph demonstrating transport network.

at s and arrival time at z, respectively. Then, the journey time of P is defined as $arr(z) - dep(s)$. The fastest path from u to v in a temporal graph is a time-respecting path that minimizes the journey time. For example, in the temporal graph depicted in Figure 1, we represent edge e_4 with the tuple $(1, 2, 4, 3)$; here vertex 1 acts as left vertex $left(e_4)$, vertex 2 as the right vertex $right(e_4)$, 4 denotes the departure time $dep(e_4)$ from vertex 1, 3 denotes the duration time from 1 to 2, and thus 7 is the $arr(e_4)$ at vertex 2. The time respected path formed by a sequence (e_8, e_{11}) of edges is the fastest path from 2 to 7 as its journey time is **7**, which is minimum when compared to the rest of the paths between the same end vertices. Now, we shall look at the formal definition of FASTEST PATH DURATION problem.

FASTEST PATH DURATION problem
Input: A temporal graph G, a source vertex s.
Output: Fastest path duration from s to the rest of the vertices in G.

The technical motivation for exploring and developing a parallel algorithm for the Fastest Path Duration problem arises from existing solutions' inherent limitations and challenges in handling massive datasets and exploiting parallelism. Notably, traditional serial algorithms like the edge stream-based Single Pass FPD [21] and TRG-based FPD algorithm [7] exhibit significant drawbacks, including overhead and lack of parallelism.

The Single Pass Fastest Path Duration algorithm encounters challenges with overhead and a lack of parallelism, which limit its efficiency and scalability. The primary source of overhead stems from maintaining dominated tuples and selecting the optimal tuple from all potential candidates in real time. Specifically, when processing an edge $\langle u, v\rangle$, the algorithm attempts to extend a partial journey by dynamically choosing the most suitable partial journey among all available options during runtime. This approach, while conceptually straightforward, incurs significant overhead due to the on-the-fly computation and decision-making, further compounded by its serial execution that fails to capitalize on the advantages of parallel processing. Similarly, the TRG algorithm fails to harness parallel computing capabilities as it involves depth first search based graph traversal and overhead of processing chain edges. Further examination of Delling et al.'s multi-core-based fastest path algorithm reveals theoretical and practical limitations [4]. Theoretically, parallelism is confined to the outgoing edges of the source vertex, resulting in limited opportunities for concurrent execution. Practically, the benefits of parallelism diminish beyond a certain threshold, with observed speedup decreases after employing approximately six threads or more. This behaviour underscores the challenges in effectively managing parallelism in real-world scenarios, particularly with extensive data sets that necessitate massive parallelism.

The advent of General-Purpose computing on Graphics Processing Units (GPGPU) offers a promising avenue to address these challenges. With their capacity for massive parallelism, GPUs present an opportunity to overcome the constraints of existing FPD algorithms. However, adapting both serial algorithms namely single-pass [21] and TRG algorithm [7] to a parallel context seems not fruitful due to their inherent sequential nature. Towards designing a GPU-based parallel algorithm for FPD, we observe that the state-of-the-art ESDG serial algorithm [19] is a good choice as it is based on the breadth first search (BFS) graph traversal, which is well explored in the parallel community. The ESDG algorithm performs BFS kind of traversal multiple times, each time starting from a different departure time. To fully leverage parallel processing capabilities, one might consider executing all BFS traversals simultaneously, in addition to applying each BFS in parallel. Although this approach aims to maximize efficiency, it can introduce redundant work, which is practically computed and shown in Table 1. In this context, repeated processing of the same edge multiple times is referred to as redundant work. Managing this redundancy is crucial for optimizing the efficiency of the parallel algorithm, ensuring that the benefits of GPU are fully realized without compromising the algorithm's performance.

Table 1: Percentage of Redundant Computations

Datasets	# Unique Computations	# Redundant Computations	Total Computations	% of Redundant Computations
Chicago	3,674	105,067	108,741	96.62
London	5,589,230	555,141,081	560,730,311	99
LosAngeles	1,206,152	61,815,191	63,021,342	98.09
Madrid	1,398,350	232,275,754	233,674,104	99.4
Newyork	18,920	267,855	286,775	93.4
Paris	958	9,003	9,960	90.38
Petersburg	1,087,595	142,150,759	143,238,355	99.24
Sweden	2,319,673	91,077,078	93,396,751	97.52
Switzerland	998,011	30,658,163	31,656,174	96.85
CollegeMsg	21,292	776,002	797,293	97.33
email	172,688	32,669,981	32,842,669	99.47
mathoverflow	36,767	483,572	520,339	92.93
stackoverflow-a2q	106,396	281,372	387,768	72.56
stackoverflow-c2a	338,465	648,396	986,860	65.7
stackoverflow-c2q	87,457	13	87,469	0.01

Computing the fastest paths in a temporal graph is challenging due to the dynamic nature of the graph. Here, the attributes of the edges, such as departure and arrival times, change over time. This dynamic aspect means that a sub-path of the fastest overall path may not be the fastest between its own endpoints. This characteristic sets temporal graphs apart from static ones and makes traditional shortest-path algorithms unsuitable. For instance, as illustrated in Figure 1, the fastest path from vertex 3 to vertex 7, utilizing edges e_6 and e_9, has a total duration of 9. However, the sub-path from vertex 3 to vertex 8 via e_6 is not the fastest because using e_5 achieves a minimum duration.

Our Contributions. Achieving more parallelism, reducing redundant computations, and effectively implementing pruning strategies are crucial to obtaining efficient parallel algorithms for FPD problem. However, balancing these elements is a challenging task. To address this, we designed three novel GPU-based parallel algorithms to solve FPD problem, namely Multiple Breadth First Search (MBFS), Local Work-lists (LW), and Level Order (LO) and implement them on a GPU architecture machine. Our key contributions are outlined as follows.

- We design a GPU-friendly data structure referred to as *level order ESD-graph*, helps to retrieve nodes appearing in a particular level and still able to access the outgoing neighbours of any node.
- We design three GPU-based parallel algorithms to solve FPD, by considering the trade-off between the key parameters parallelism, redundancy, and pruning.
- We also present proof of correctness for the devised algorithms.
- We analyze the behaviour of our parallel algorithms experimentally, by running on various types of real world datasets and obtain key insights.

2 PRELIMINARIES

In this section, we briefly describe three state-of-the-art serial algorithms to solve FPD problem, which are useful as baseline algorithms. Later, we provide the functionality of a few atomic operations, which are useful in our GPU based parallel algorithms.

The single pass algorithm receives a temporal graph in edge stream format, in which the edges are ordered in increasing order based on their departure time. In the case of TRG-algorithm and ESDG-algorithm, a temporal graph is converted to appropriate transformation graphs during the preprocessing time, and the main algorithms perform necessary traversal from a source vertex during query time.

2.1 Single-Pass Algorithm

Wu et al., [21] designed a serial algorithm to solve the FPD problem using edge stream of G. This algorithm keeps track of certain important paths called dominating paths. For each vertex v, there is a L_v of pairs, where each pair captures the departure and arrival times of a dominating path from a source vertex to v. For each edge $e = (u, v, t, \lambda)$, the algorithm determines if a dominating path from a source vertex s to vertex u can be extended to reach v through e. If an extension is possible, the algorithm selects a pair from L_u that corresponds to the latest departure path from s to u and arrives on or before time t, which minimizes the duration of the path reaching v. The next step involves updating the list L_v consisting of dominating tuples. If L_v already contains a pair with the same start time, the algorithm updates the arrival time in the same pair in L_v to the new, smaller value computed if it represents an improvement. If no such pair exists, the algorithm inserts the new pair into L_v. Subsequently, any dominated elements in L_v are removed to retain only the most efficient paths. Throughout this process, $f[v]$ tracks the duration of the fastest path from s to v. If the minimum duration represented by $f[v]$ changes, the algorithm updates $f[v]$ to reflect the minimum journey duration seen so far. This systematic approach ensures the algorithm accurately identifies and updates the fastest duration.

2.2 TRG-Algorithm

Gheibi et al., [6, 7] proposed an algorithm to solve FPD problem using Time Respecting Graph (TRG), a transformed graph of G. The departure events of a temporal graph are treated as nodes in the TRG graph. Also, an arrival event at a vertex is considered as a node in the TRG graph, if there is no outgoing edge after such event. For each edge (u, v, t, λ) in G, an edge is added in the transformed graph from the departure event (u, t) to an appropriate event at v, such that its departure or arrival time is at least $t + \lambda$. The algorithm sorts nodes by time, performs depth first search from nodes corresponding to departure events of a source vertex, and explores unvisited nodes within the time constraints. It initializes the DFS stack, sets start times for paths, and updates distances if a faster path is found. Unvisited neighbours meeting the time constraints are explored, and the process continues for time respecting chain neighbours.

2.3 ESDG Algorithm

Ni et al., [18] have transformed a temporal graph to an equivalent Edge Scan Dependency (ESD) graph for solving the earliest arrival path problem in temporal graphs. Let G be a temporal graph and $\tilde{G}$ be an ESD-graph of G that is defined based on the following constraints. For every edge e in G, there is a node v_e in $\tilde{G}$. Further, for any two edges $e = (u, v, t, \lambda)$, $f = (v, w, t', \lambda')$ in G, we add an edge from v_e and v_f, if $t' \geq t + \lambda$, and there does not any exist any edge g from v to w, such that departure time of g is at least $t + \lambda$ and arrival time of g is less than $t' + \lambda'$. As each node v_e in $\tilde{G}$ corresponds to an edge e in G, we maintain 4 attributes (u, v, t, a) at every node, where u, v, t, and $a = t + \lambda$, denotes the left vertex, right vertex, departure time and arrival time of e, respectively. An example of an ESD-graph is shown in Figure 2. As each node in $\tilde{G}$ corresponds to an edge in G, the *id* of each edge is associated with the corresponding node, and is shown in bold. Although, the size of $\tilde{G}$ can be more than the size of G, we explore parallelism and pruning to avoid the issues raised due to the increase in size. For a node x in an ESDG $\tilde{G}$, if the node x has no parents, then $level(x)$ is set to 1; otherwise $level(x)$ is set to $1 + \max_{y \in par(x)} \{level(y)\}$, where $par(x)$ denotes the set of parent nodes of x in $\tilde{G}$. It effectively assigns a level of 1 to nodes without parents and for all other nodes, their level is one more than the maximum level among their parents. For any two consecutive edges in any time-respecting path in G, the departure times are in monotonically increasing order, as the duration time on every edge is a positive integer. Thus the ESD graph is a directed acyclic graph.

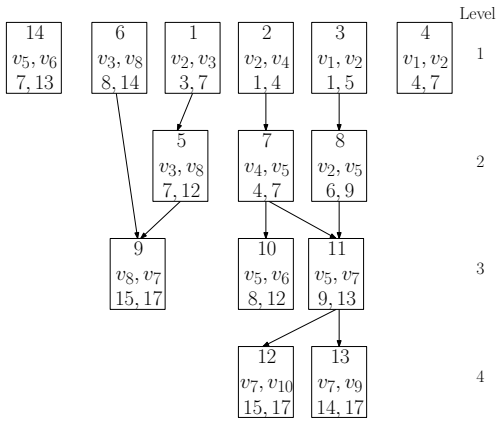

Figure 2: Edge Scan Dependency Graph

Corollary 2.1. *[19, Corollary 3.4] Let P be the fastest path from a vertex s to a vertex z in a temporal graph G. Then there exists a useful dominating path Q from s to z such that $dep(Q) = dep(P)$ and $arr(Q) = arr(P)$.*

Lemma 2.2. *[19, lemma 3.5] A sequence $e_1, \ldots, e_k$ of k edges in G is a useful dominating path if and only if a sequence $v_{e_1}, \ldots, v_{e_k}$ of k vertices in $\tilde{G}$ is a path. Further, the journey time of the path from e_1 to e_{k-1} in G is equal to the journey time of the path from v_{e_1} to $v_{e_{k-1}}$ in $\tilde{G}$.*

Srikanth et.al., [19] have proposed a serial algorithm to solve FPD problem and introduced the notion of useful dominating paths. They have shown that all the useful dominating paths of G are preserved in $\tilde{G}$. In other words, there is a one-to-one mapping between useful dominating paths in G and paths in $\tilde{G}$ as shown in Lemma 2.2. From Corollary 2.1, every fastest path in a temporal graph is a useful dominating path. Due to this, processing all paths in $\tilde{G}$ is sufficient to compute the fastest path duration of all vertices in G. Given a source vertex s, this algorithm iterates over all the source nodes in $\tilde{G}$ corresponding to the outgoing edges of s in decreasing order based on their departure time. In each iteration, all the unvisited reachable nodes are identified, journey times are computed for all such nodes, and maintain the minimum journey time at each vertex.

2.4 Atomic Operations

We use the following atomic functions that are supported in CUDA in our parallel algorithms to ensure the correctness while updating the same variable by multiple threads. The atomic operations `atomicMin()` and `atomicMax()` are CUDA atomic functions that operate on shared or global memory, ensuring that when multiple threads attempt to update a variable with the

Table 2: Frequently Used Notations

Symbol	Description
G	A temporal graph
$\tilde{G}$	An Edge Scan Dependency Graph of G
n, m	Number of vertices & edges in G
$\tilde{n}, \tilde{m}$	Number of vertices & edges in $\tilde{G}$
$x : (id, u, v, t, a)$	node in $\tilde{G}$
$left(x), right(x)$	left and right vertices of a node x in $\tilde{G}$
$dep(x)$	departure time of edge x in G
$arr(x)$	arrival time of edge x in $G = dep(x) + \lambda(x)$
l_{max}	maximum no. of nodes in a level in $\tilde{G}$
d_{max}	maximum out-degree of a node in $\tilde{G}$
l	no. of levels in $\tilde{G}$

minimum or maximum of their values, the operation is performed atomically. This guarantees that the final value stored is indeed the minimum or maximum of all values proposed by the threads.

The *atomic compare-and-swap* function (`atomicCAS`) performs a compare-and-swap operation atomically, meaning it compares the content of a given memory location to a given value and, only if they are equal, modifies the contents of that memory location to a new given value. For instance, the function $atomicCAS(address, x, y)$ receives 3 parameters, where address refers to a memory location, x is the value to compare with the value at the specified address and y is the new value to write to the specified address. This function always returns the old content referred to by address.

3 EFFICIENT PARALLEL ALGORITHMS FOR FASTEST PATH DURATION

The running time of a parallel algorithm depends on many factors such as the amount of serial portion, the number of data items involved in pruning, and the number of redundant computations. Further, many of these factors depend on each other. For instance, increasing the parallelism leads to more redundant work as shown in Table 1. Similarly, pruning of unreachable nodes in a graph requires explicit maintenance of a worklist of active nodes, etc. We explore various design choices systematically and present multiple novel algorithms to solve FDP, along with their merits and demerits.

We use an ESD-graph $\tilde{G}$ of a temporal graph G in our algorithms. Without loss of generality, we assume that $\tilde{G}$ is constructed in preprocessing time as it is independent of a source vertex and is required to construct exactly once. The key idea common to three of our proposed algorithms is to go through various source nodes in a ESD-graph corresponding to the outgoing edges of a source vertex and identify the reachable nodes to compute the journey times of all paths originating from source nodes.

Notation. A node in $\tilde{G}$ that corresponds to an outgoing edge of a source vertex in G is referred to as *source node*. For each vertex v in G, without loss of generality, we assume that the departure times of all the outgoing edges of v are unique in our algorithm. Later, we relax this in our implementation. The frequently used notations in this paper are shown in Table 2 for a quick reference.

Initialization. Given an ESDG ($\tilde{G}$) of a temporal graph G, and a source vertex s, all three algorithms commence with an initialization phase. During this phase, $journey[v]$ is set to ∞ for all vertices v in the vertex set $V(G)$ excluding the source vertex s, for which $journey[s]$ is set to 0. This initialization sets the stage for finding the fastest journey from the source vertex s to every other vertex in graph G.

3.1 GPU-based parallel algorithm - Multiple Breadth-First Search

The key idea in this algorithm is to go through relevant source nodes in the non-increasing order of their departure times and apply worklist-based parallel BFS from each node, to identify reachable nodes.

Algorithm 1: Multiple Breadth-First Search

Input: A source vertex s, and an ESDG $\tilde{G}$ of a temporal graph G.
Output: For each vertex z in G, $journey[z]$ stores the fastest duration from s to z.

```
1  for each node z in G̃ do status[z] = false ;
2  for each node z in G̃ such that left(z) == s in non-increasing order
     of dep(z) do
3      status[z] = true; frontier = ∅; frontier.insert(z);
4      while frontier ≠ ∅ do
5          newFrontier = ∅;
6          for each node x in frontier in parallel do
7              journey[right(x)] =
                 atomicMin{journey[right(v)], arr(x) − dep(z)};
8              for each neighbor y of x in G̃ do
9                  if atomicCAS(status[y], false, true) == false then
                     newFrontier.insert(y) ;
10         frontier = newFrontier;
```

In the initial step, we set the $status$ array to $false$ for each node in $\tilde{G}$, and update to $true$ for a particular node when it is visited for the first time. The algorithm begins with an outermost for loop (Line 2), to process source nodes z, where $left(z) = s$, sorted in descending order of their departure times. Upon entering the loop, each node z is marked as visited. The BFS starts with a $frontier$ containing a node z and progresses through a while loop, where nodes in the current $frontier$ are processed in parallel and their unvisited neighbors are added to $newFrontier$, which then serves as the next iteration's $frontier$. For each reachable node x from a source node z, we compute the journey time as $arr(x) - dep(z)$, and atomically update the $journey[right[x]]$ in Line 7, if the new duration is lesser than the existing one. Due to the descending departure time order, we ignore already visited nodes since they cannot offer a shorter journey time. This is achieved in Line 9, with the help of `atomicCAS` instruction. Now, we shall look at the loop invariant of this algorithm. After i^{th} iteration of Algorithm is over, $journey[z]$ maintains the journey time of the fastest path among all the paths from s to z such that their departure time is on or after d_i, where d_i is the departure time of the i^{th} source node as shown per Line 2. The correctness of the algorithm follows from the proposed loop invariant. The outermost loop runs for $d_{out}(s)$ times, where $d_{out}(s)$ denotes the out-degree of a source vertex s, with unique departure times in the underlying temporal graph. In each phase, we perform parallel BFS whose depth is at most l, where l denotes the number of levels in $\tilde{G}$. Further, the outgoing edges of every node are processed sequentially due to node-centric parallelism. As shown in Line 10, the next frontier gets created from the current frontier atomically and this takes $O(l_{max})$ time, where l_{max} denotes the maximum number of nodes in a level. Thus the overall depth of this algorithm is bounded by $O(d_{max} * l * (d_{max} + l_{max}))$, where d_{max} denotes the maximum degree of a node in $\tilde{G}$. Rather than preparing the next frontier atomically, a prefix-sum-based scan operation can reduce the depth. In such case, l_{max} in the proposed depth reduces to $\log l_{max}$. As all the nodes and edges in $\tilde{G}$ are processed at most once, the work is bounded by $O(\tilde{n} + \tilde{m})$.

Applying Algorithm 1 to the ESD-graph depicted in Figure 2, with v_2 as the source vertex, results in a sequential exploration across multiple phases. In the first phase, nodes 8, 11, 12, and 13 are reachable from 8. Subsequently, the next phase starts at node 1 and processes nodes 5 and 9. The last phase explores the nodes 2, 7, and 10. Notably, vertices 11, 12, and 13 are not revisited in Phase 3, as these are explored in Phase 1.

Our algorithm does not process the nodes that are not reachable from source nodes and thus performs pruning effectively. Also, no node is processed multiple times, and hence redundant work is avoided. However, this algorithm suffers from serial processing across multiple breadth-first search phases.

Challenge. How to overcome the serial processing across multiple parallel BFS phases?

3.2 GPU-based parallel algorithm - Local Worklist

Each BFS phase in the previous algorithm maintains a local worklist for every level. In contrast, the main theme in this algorithm is to maintain a single local worklist for each level to keep the active nodes that are reachable from source nodes. Another key ingredient in this algorithm is to propagate the departure time of each source node to the respective reachable nodes by applying single-level order traversal rather than multiple BFS traversals. This helps to avoid the kind of serial processing incurred in the earlier algorithm.

Algorithm 2: Local Worklist Algorithm

Input: A source vertex s, and an ESDG $\tilde{G}$ of a temporal graph G
Output: For each vertex z in G, $journey[z]$ stores the fastest duration from s to z.

1 **for** *each node* x *in* $\tilde{G}$ **do**
2 $\quad startTime[x] = -\infty$; $status[x] = false$;
3 **for** *each node* x *in* $\tilde{G}$ *such that* $left(x) == s$ **do**
4 $\quad startTime[x] = dep(x)$; $wList[level(x)].insert(x)$;
5 **for** *each level* l *from* 1 *to* L **do**
6 $\quad$ **for** *each node* x *in* $wList[l]$ **in parallel do**
7 $\quad\quad journey[right(x)] =$ atomicMin$\{journey[right(v)], arr(x) - startTime[x]\}$;
8 $\quad\quad$ **for** *each neighbor* y *of* x *in* $\tilde{G}$ **do**
9 $\quad\quad\quad startTime[y] =$ atomicMax$\{startTime[y], startTime[x]\}$;
10 $\quad\quad\quad$ **if** atomicCAS*(status[y], false, true) == false* **then** $wList[level(y)]$.insert(y) ;

We describe Algorithm 2 as follows. For each level i in ESDG we have a local worklist denoted by $wList[i]$ that helps to maintain active nodes in level i. For each node x in $\tilde{G}$, we use $startTime[x]$ to keep track of the maximum departure time of all paths starting from a source node and ending at x. At the beginning of the algorithm, we initialize $startTime$ and $status$ of all nodes in $\tilde{G}$ as $-\infty$ and $false$, respectively. Then for all nodes x such that $left(x) = s$, $startTime$ is assigned as $dep(x)$ and these nodes are inserted into worklists of their corresponding level. The outermost for loop (Line 5) of the algorithm is responsible for processing the levels one at a time. At i^{th}-iteration, all nodes x in $wList[i]$ are processed in parallel and the journey times of paths ending at $right[x]$ is updated using `atomicMin` instruction in Line 7. Then $startTime$ of x is propagated to all its neighbors. A node y can receive multiple starting times, whereas the arrival time of y is fixed. Therefore, we capture the largest starting time in Line 9, to compute the fastest journey using `atomicMax`. Then all its neighbors not yet added to any worklist are added to their corresponding worklist in Line 10, using `atomicCAS` instruction. At the end of this algorithm, we have the fastest journey to all the nodes from s. The correctness of this algorithm is shown in Section 3.4. We use k to denote the number of sequential insertions across all the worklists. Let s^i_j denote number of dependencies from i^{th} level to j^{th} level. Then, k is $\sum_{i=1}^{l} \max\{s^i_{i+1}, s^i_{i+2}, \ldots, s^i_l\}$). The depth and work of this algorithm are $O(l+k)$ and $O(\tilde{m}+\tilde{n})$ respectively. We can avoid atomically inserting nodes in the appropriate worklists, by delaying all the insertions until the algorithm reaches the corresponding level. Then the prefix sum technique can be used to calculate the necessary indices and insert the nodes in parallel. The depth of the resultant algorithm is $O(l*(d_{max}+\log\alpha)+\alpha)$, where $\alpha = \max_{1<j\le l} \sum_{i=1}^{j-1} s^i_j$, which denotes the maximum number of incoming edges to a level in $\tilde{G}$.

Applying Algorithm 2 to the ESD-graph depicted in Figure 2, with v_2 as the source vertex, it identifies the corresponding ESDG nodes as 1, 2, and 8. Nodes 1 and 2, appearing at level 1, are allocated to the level 1 worklist, while node 8, found at level 2, is placed in the level 2 worklist. Worklists for levels 3 and 4 remain empty. The algorithm then concurrently processes the nodes in a worklist and moves to the next level worklist, ensuring an organized and efficient graph traversal.

Each node in the ESD-graph is processed at most once and unreachable nodes from source nodes are not processed, thus Algorithm 2 ensures not to be involved in redundant work and performs pruning efficiently. Another strength of this algorithm is that it has a single phase due to level order traversal and avoids serial processing across multiple phases. Despite these benefits, this algorithm experiences serial processing and overhead due to the creation of worklists containing active nodes.

Challenge. How to avoid the overhead and serial processing while creating dynamic worklists?

3.3 GPU-based parallel algorithm - Level-order Traversal

At a high level, we apply level order traversal on levels of ESD-graph rather than applying on a worklist of active nodes. This technique helps to avoid the creation of worklists explicitly.

Algorithm 3: Level-order Traversal

Input: A source vertex s, and an ESDG $\tilde{G}$ of a temporal graph G
Output: For each vertex z in G, $journey[z]$ stores the fastest duration from s to z.

1 **for** *each node* x *in* $\tilde{G}$ **do** $startTime[x] = -1$; $status[x] = false$;
2 **for** *each node* x *in* $\tilde{G}$ *such that* $left(x) == s$ **do** $startTime[x] = dep(x)$; $status[x] = true$;
3 **for** *each level* l *from* 1 *to* L **do**
4 $\quad$ **for** *each node* x *in level* l **in parallel do**
5 $\quad\quad$ **if** $status[x] == true$ **then**
6 $\quad\quad\quad journey[right(x)] =$ atomicMin$(journey[right(x)], arr(x) - startTime[x])$;
7 $\quad\quad\quad$ **for** *each neighbor* y *of* x *in* $\tilde{G}$ **do**
8 $\quad\quad\quad\quad atomicMax(startTime[y], startTime[x])$;
9 $\quad\quad\quad\quad status[y] = true$;

Now, we describe Algorithm 3, in which the initialization steps and the core computations are the same as the previous algorithm, except for

maintaining explicit worklists. All the source nodes in the respective levels of $\tilde{G}$ are active at the beginning. For each level, the number of threads that we launch equals to the number of nodes in the respective level. During each iteration, i^{th}-thread considers i^{th} node in the respective level of $\tilde{G}$ and performs necessary operations if the node is active. The main operations include the computation of the journey time, propagation of the starting time to its neighbors, and marking the neighbors as active, as shown in Lines 6, 8, and 9. For readability purposes, vertex-centric parallelism is followed in Algorithm 3, and thus the depth and work of the algorithm are $O(l * d_{max})$ and $O(\tilde{m} + \tilde{n})$, respectively. In our next variant, we follow edge-centric parallelism instead of vertex-centric parallelism to enhance parallelism and reduce the depth. The resultant algorithm has a depth of $O(l)$ and requires $O(\tilde{m})$ work.

Although pruning is not followed in Algorithm 3, each node in the ESD-graph is processed exactly once due to the level order traversal. Thus the algorithm avoids redundant work. Moreover, there is no overhead in this algorithm, as the creation of worklists is avoided. Further, due to edge-centric parallelism, only the number of levels in $\tilde{G}$, limits the parallelism.

3.4 Algorithm Correctness

In this section, we turn our attention to prove the correctness of Algorithm 2 and Algorithm 3. Algorithm 3 processes all nodes in a level concurrently, whereas Algorithm 2 processes all nodes in a local worklist associated with a level, in parallel. All the levels in an ESD-graph are processed iteratively in both algorithms. Thus, we use induction on the number of levels in $\tilde{G}$, to prove the correctness of both algorithms in Lemma 3.1 and Theorem 3.2.

LEMMA 3.1. *Let $\tilde{G}$ be an ESDG of a temporal graph G and s be a source vertex in G. Let $P = (x, y)$ be a path in $\tilde{G}$ such that, $left(x) = s$, $right(y) = z$, $1 \leq level(x)$, $level(y) \leq k$. At the end of k^{th} iteration of Algorithm-3, the journey time of P is computed.*

PROOF. We prove this lemma using induction on k. The way of initializing the journey times of all the vertices in the initialization phase establishes the base case of the induction. Utilizing the induction hypothesis, by the end of processing vertices in level $k-1$, we assume that the journey times of all paths that end at level k or earlier are accurately computed. From the premise of this lemma, we have a path P from x to y, where $left(x) = s$, $right(y) = z$, $1 \leq level(x)$, $level(y) \leq k$. The path P can fall in one of the following three scenarios based on the levels of x and y. If $1 \leq level(x), level(y) < k$, as per our induction hypothesis, the claim is true. Let us consider the second case in which, $1 \leq level(x) < k$ and $level(y) = k$. This ensures that there is a predecessor y' of y in P, and the journey time from x to y' is computed due to induction hypothesis. Also, due to Line 3 in Algorithm 3, the status of the outgoing neighbour y of y' must have become true. In other words, the vertex y that appear in level k is active at the end of processing level $k-1$. During the k^{th} iteration, the node y is processed and hence the claim is true. Now, we are left with third case in which, $level(x) = level(y) = k$. These kind of nodes become active in the initialization phase and are processed in k^{th} iteration. By systematically processing all levels, the algorithm guarantees that the journey time of P is computed, at the end of k^{th} iteration. □

THEOREM 3.2. *Given a temporal graph G and a source vertex s, upon completion of Algorithm 3, the value of $journey[z]$ for each vertex z in G represents the journey time of a fastest path from the source vertex s to z.*

PROOF. Let P be a fastest path from s to z. From Corollary 2.1 and Lemma 2.2, there exists a path Q in $\tilde{G}$, such that $dep(P) = dep(Q)$ and $arr(P) = arr(Q)$. Lemma 3.1 ensures that the journey times of all paths including Q are computed. Line 6 of Algorithm 3 compares the journey times of all paths ending at a node x, such that $right[x] = z$ and stores the minimum journey time in $journey[z]$. Thus the theorem holds true.

□

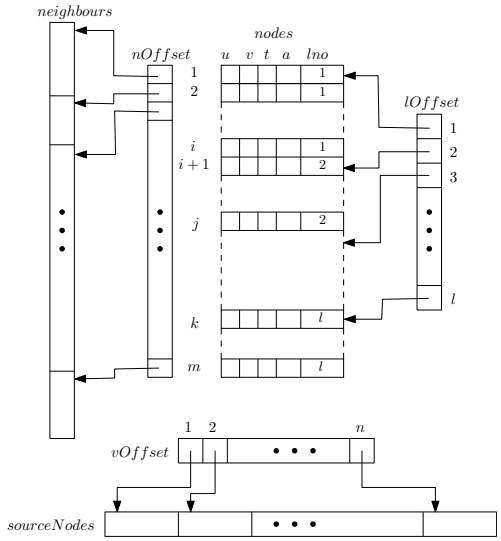

Figure 3: Level Order ESD-graph Data structure

4 IMPLEMENTATION DETAILS

We start this section with the representation of *Level Order ESD-graph*. Later, we present various implementation details such as creating worklists, handling duplicates and worklists with full capacity, etc. to reduce the practical running time.

Level Order ESD-graph. The nodes in an ESD-graph are arranged in level order traversal in [21] and topological structure is not used. In contrast, the topological structure is used in [19], rather than arranging nodes in a specific order. We propose a new representation LEVEL ORDER ESD-graph by combining the merits of both representations. This data structure consists of six arrays that help to retrieve necessary nodes in an ESD-graph efficiently, and its pictorial representation is shown in Figure 3.

- $nOffset$ and $neighbours$ arrays help to store the topological structure of $\tilde{G}$ as shown in left arrays in Figure 3. For each node x in $\tilde{G}$, neighbouring nodes of x start from $neighbours[nOffset[x]]$.
- $lOffset$ and $nodes$ arrays are useful to retrieve nodes appearing at a particular level. For each level i in $\tilde{G}$, nodes appearing in level i start from $nodes[lOffset[i]]$.
- For each source vertex u in G, the source nodes in $\tilde{G}$ can be retrieved from $sourceNodes[vOffset[u]]$.

Memory Coalescing. For GPU friendly memory coalesced accessing, the ESDG nodes in the graph are relabeled such that all the nodes in any level appear consecutively. Moreover, these are arranged in increasing order of level number as shown in Figure 4. We obtain the topological structure of an ESD-graph using a preprocessing algorithm proposed in [18], and apply a variant of the topological sorting algorithm to obtain level numbers to all the nodes in $\tilde{G}$.

Prefix sum and atomicAdd. In Algorithm 1, the number of threads for each kernel equals the number of nodes in the current frontier. While creating the next level frontier of nodes, the prefix sum technique is preferable than `atomicAdd` inserts in theory. However, the prefix sum technique is expensive in practice, as it needs an extra kernel launch. This is not worthwhile as the frontier size is normally small and only occasionally reaches the order of 1000. While appending nodes to the next level frontier using $atomicAdd$, instead of appending one node at a time, we reserve the necessary space sufficient to append all the nodes to be inserted by a particular thread. This optimization minimizes the atomic operations and retains duplicates. As the removal of duplicates is costly, it is not worthwhile. Although we

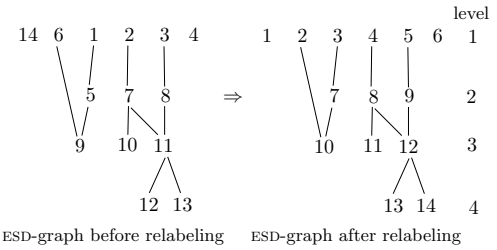

Figure 4: Nodes are relabeled level-wise

have duplicates in the current level, for any vertex in the current level, their outgoing nodes are inserted only once in the next level. This avoids cumulative effect. With this implementation, we obtain better running times but at the cost of more memory.

Worklist Allocation. For Algorithm 2, we allocate worklists for each level with a size equal to the total number of nodes in that level, representing the maximum size the worklist can reach. These worklists are allocated as single long arrays, enabling us to target any level's worklist and insert nodes. To insert a node into a worklist, we use *atomicAdd* to the end index and then insert, ensuring sequential pushing to the worklist. Insertions across different worklists happen in parallel but are sequential if specific insertions target the same worklist.

Full Worklist. In Algorithm 2, there is an interesting fact to observe. When we reach a level at which the whole worklist is full, then from then onwards worklists will be full by the time we reach there. So whenever we encounter a worklist at its full capacity, we switch to Algorithm 3 because the next level worklist gets full anyway, and thus it is better to process all the nodes of the following levels rather than using worklists. We use this optimization to avoid creating certain worklists and obtain significant improvement in the running times.

In Algorithm 3 for level-order traversal, we launch as many threads as the number of nodes in that level, ensuring consecutive node numbering for efficient memory coalescing. This approach is more cache-efficient than the local worklist approach, as it avoids inserting nodes into worklists and changing their status from $false$ to $true$. Even if multiple threads attempt to flip the status flag, there are no issues, as every thread performs the same action.

Vertex Centric and Edge Centric Parallelism. We implement Algorithm 3 using vertex-centric parallelism as well as edge-centric parallelism independently. For the vertex-centric approach, all the nodes in $\tilde{G}$ are sorted level-wise and we store the indices of the first nodes from each level in another offset array. This helps to process only nodes appearing in a particular level. Similarly, for the edge-centric approach, we sort the outgoing edges in $\tilde{G}$ level-wise in preprocessing time.

5 EXPERIMENTS

In this section, we start with our experimental setup and describe the characteristics of various data sets. Later, we compare the running times of our GPU based parallel algorithms with the state-of-the-art algorithms. We then identify various parameters of temporal graphs and their ESD-graphs that influence the practical performance of our algorithms.

5.1 Experimental setup

The experimental setup includes an AMD EPYC 7742 64-core processor running at frequencies ranging from 1.5 GHz to 3.39 GHz, with a 32 GB RAM. This system is complemented by an NVIDIA A100-SXM4-40GB GPU, featuring CUDA driver version 11.4 and runtime version 11.4. The GPU offers 39.59 gigabytes of global memory and operates at a GPU clock rate of 1.41 GHz, with a memory clock rate of 1.21 GHz. The GPU also includes 65536 bytes of total constant memory, 49152 bytes of shared memory per block, and supports 65536 registers per block, with a warp size of 32.

Table 3: Data Set Characteristics

Dataset	n	$m = \tilde{n}$	$\tilde{m}$	l
Chicago	240	14.7K	44.9K	680
London	20.8K	8.8M	12.1M	2055
LosAngeles	13.9K	1.7M	2.3M	1910
Madrid	4.6K	1.8M	2.7M	1574
Newyork	987	412.3K	499.7M	708
Paris	411	6.5K	50.9M	93
Petersburg	7.5K	2.3M	6M	1641
Sweden	45.7K	3.9M	12.1M	2205
Switzerland	29.8K	3.9M	12.1M	1881
CollegeMsg	1.9K	59.7K	1.2M	3206
email	1K	326.9K	12.8M	5662
mathoverflow	88.5K	506.5K	65.9M	7429
stackoverflow-a2q	6M	17.8M	479.3M	1093
stackoverflow-c2a	6M	16.5M	2.1G	40468
stackoverflow-c2q	6M	20.2M	422.7M	3023

5.2 Data set Characteristics

We conducted our experiments using two types of real-time datasets, namely, Public Transport Networks and Social Networks. The public transport datasets include data from cities such as Chicago, London, Los Angeles, Madrid, New York, Paris, Petersburg, Sweden, and Switzerland. The social network datasets include data from platforms such as College Message, Email, Math Overflow, and Stack Overflow. The statistics of the datasets are shown in Table 3. The attributes of all these datasets include the number of vertices (n) and edges (m) in a temporal graph. Also, the table includes the number of nodes ($\tilde{n}$), edges ($\tilde{m}$) and the number of levels (l) in a ESD-graph.

5.3 Comparison of the proposed algorithms with the state-of-the-art algorithms

We implemented CPU-based state-of-the-art algorithms in C++ and proposed GPU-based algorithms in C++ using CUDA. We generated 100 queries for each dataset by randomly selecting source vertices from 0 to $|v|$. We then ran both the state-of-the-art and proposed algorithms using these dataset queries. We further computed the running time for each query and reported the average running time across all 100 queries. The state-of-the-art and the proposed parallel algorithms were rigorously validated to ensure correctness by comparing the results for each query against the expected outputs. The experiments were conducted in the environment described in Section 5.1. Table 4 presents the average running times per query for the three state-of-the-art algorithms described in the Section 2, including single-pass algorithm [21], TRG algorithm [7] and ESD-graph algorithm [19]. We also describe the running times of three of our parallel GPU based algorithms Algorithm 1 (MBFS), Algorithm 2 (LW), and Algorithm 3 (LOT) measured in milliseconds.

We choose a best-performing algorithm from the proposed algorithms and a best-performing algorithm from the state-of-the-art algorithms and compare their running times for each dataset to obtain a speedup column, shown in Table 4. For the very small graphs, the serial algorithms perform better because of the GPU overheads. But on large graphs, as shown in Figure 5, our algorithms achieve the average speedup of **165**× and the maximum speedup of **1383**× over the state-of-the-art algorithms. We could not construct TRG-graphs for two datasets in the preprocessing time, as the number of chain edges required for the transformation is huge.

We now highlight the crucial reasons that help to achieve better performance in our algorithms. The most important merit of MBFS and LW algorithms is pruning. As shown in Table 5, we prune from 32% to 99%

Table 4: Running Time-taken in milliseconds

	State of the art Algorithms			Proposed Algorithms			
Dataset	SP [21]	TRG [7]	ESDG [19]	MBFS	LW	LO	speedup
Chicago	7	1.3	2.36	29.6	13.4	4.1	0.32
London	4992.2	1912.7	30929	1060.8	29.2	18.6	102.83
LosAngeles	503.1	303.2	3798	529.4	18.3	15.6	19.44
Madrid	685.4	335	514.7	728.4	13.8	12.7	26.38
Newyork	30.5	12.4	4.42	188.7	11.7	3.9	1.13
Paris	52.6	4	0.28	2.8	2	0.5	0.56
Petersburg	585.9	221.3	2687	430.2	35.3	13.3	16.64
Sweden	924.4	617.2	42010	169.7	49.7	19	32.48
Switzerland	637.6	287.9	20955	132.1	52.5	15.8	18.22
CollegeMsg	4.7	4.7	1117	15.3	38.7	18.9	0.42
email	44.3	26	24477	144.3	150.6	40.6	0.64
mathoverflow	27.9	15.7	33015	5.8	169.8	47.9	6.28
stackoverflow-a2q	1521.5	-	2497.81	2.9	16.7	9.7	1,383.18
stackoverflow-c2a	2505.99	-	4398	9.8	770.5	265.5	368.25
stackoverflow-c2q	1984.04	409	11405	2.2	49.6	21.8	511.25

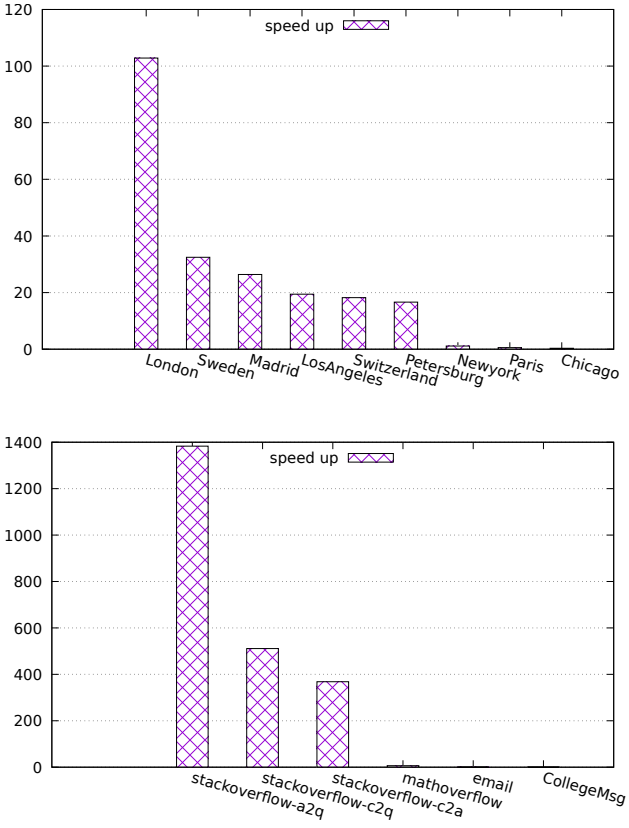

Figure 5: Speedup: For a dataset the minimum running time out of all baseline algorithms (SP,TRG,ESDG) / minimum running time out of all our algorithms (MBFS,LW,LO)

of nodes on the real-world datasets. The algorithms LW and LO achieve better performance due to the nature of single pass level order traversal, rather than multiple passes. Lastly, avoiding redundant work and boosting parallelism help improve the performance of the three proposed algorithms, as they eventually reduce the parallel depth and parallel work.

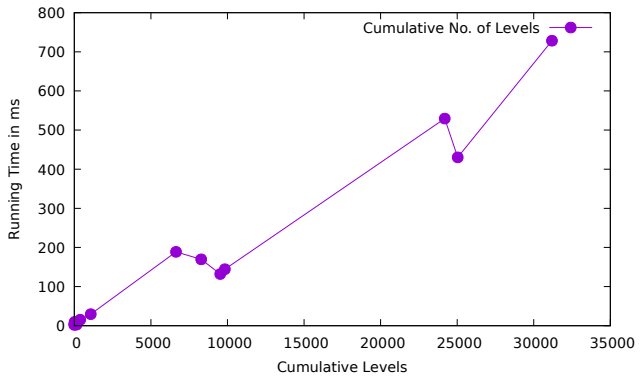

Figure 6: Cumulative levels vs Running Time of Algorithm 1

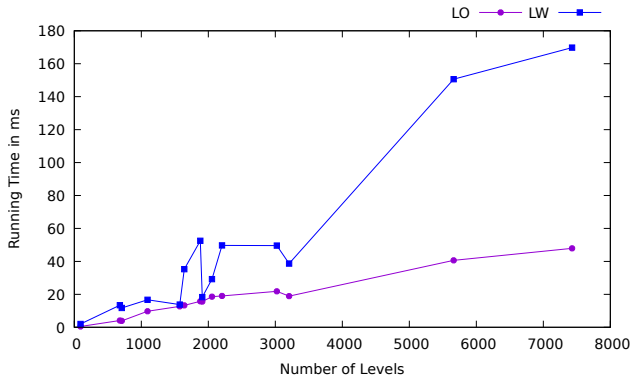

Figure 7: Number of levels vs Running Time of Algorithm 2 and Algorithm 3

5.4 Practical Performance and Parameters

Impact of cumulative levels on Algorithm 1.

In the context of Multiple Breadth-First Searches (MBFS), we first identify the average number of levels traversed in a BFS-graph traversal. From this, we then compute the cumulative number of levels traversed over all BFS-traversals. The practical running time of MBFS algorithm is proportional to the cumulative number of levels, as shown in Figure 6.

Impact of cumulative insertions on Algorithm 2. In Algorithm 2, the running time depends on the cumulative insertions. At any level, all the nodes in that worklist start to insert their neighbours in their corresponding level. These insertions across different worklists happen in parallel, but those at the same level happen in serial. The total number of serial insertions across all the worklists is referred to as *cumulative insertions*. The number of levels in $\tilde{G}$ as well as the *cummulative insertions* are a few parameters that could influence the performance of Algorithm 2. Our experimental analysis reveals that the running time highly correlates with the number of levels in $\tilde{G}$, than the cumulative insertions, which is shown in Figure 7.

Impact of the number of levels on Algorithm 3. The level order traversal's performance is significantly influenced by the number of levels within the ESD-graph. Essentially, an increase in the number of levels directly extends the algorithm's running time. The parameter *the number of levels* essentially serves as a metric to measure the practical depth of our algorithm. Illustrations in Figure 7 elucidate the relationship between the running time of Algorithm 3 (LO) and the number of levels in $\tilde{G}$.

Table 5: Correlation of Parameters on MBFS, LW and LO Running Times

Dataset	Average Out Degree (1)	Avg #levels Each Phase (2)	Pruning (%)	#Levels	Cumulative Levels (1)*(2)/100	Cumulative Insertions	MBFS	LW	LO
Paris	17	841	89.1	93	142.97	281	2.8	2	**0.5**
Chicago	62	1731	75.3	680	1073.22	1323	29.6	13.4	**4.1**
Newyork	422	1573	95.4	708	6638.06	15647	188.7	11.7	**3.9**
Madrid	402	7758	23.3	1,574	31187.16	179813	728.4	13.8	**12.7**
Petersburg	315	7948	44.4	1,641	25036.2	502294	430.2	35.3	**13.3**
Switzerland	137	6954	74.2	1,881	9526.98	339982	132.1	52.5	**15.8**
LosAngeles	130	18606	32.1	1,910	24187.8	190338	529.4	18.3	**15.6**
London	453	12636	37.3	2,055	57241.08	1261738	1060.8	29.2	**18.6**
Sweden	89	9308	41.5	2,205	8284.12	520584	169.7	49.7	**19**
email	324	3035	48.4	5,662	9833.4	834	144.3	150.6	**40.6**
stackoverflow-a2q	3	91	99.8	1,093	2.73	14145	**2.9**	16.7	9.7
stackoverflow-c2q	4	198	99.8	3,023	7.92	21865	**2.2**	49.6	21.8
CollegeMsg	31	1236	70	3,206	383.16	278	**15.3**	38.7	18.9
mathoverflow	6	215	94.9	7,429	12.9	140	**5.8**	169.8	47.9
stackoverflow-c2a	4	568	98.6	40,468	22.72	1622	**9.8**	770.5	265.5

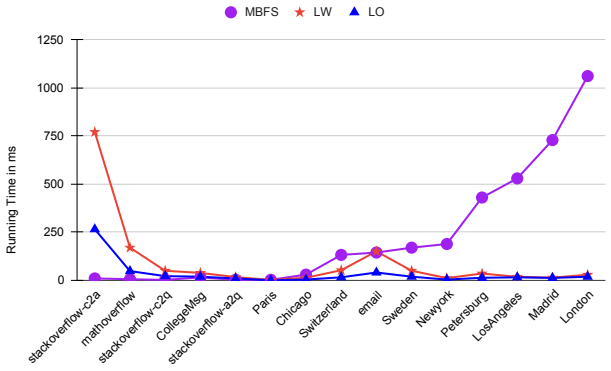

Figure 8: Running times of Proposed Algorithms MBFS, LW and LO

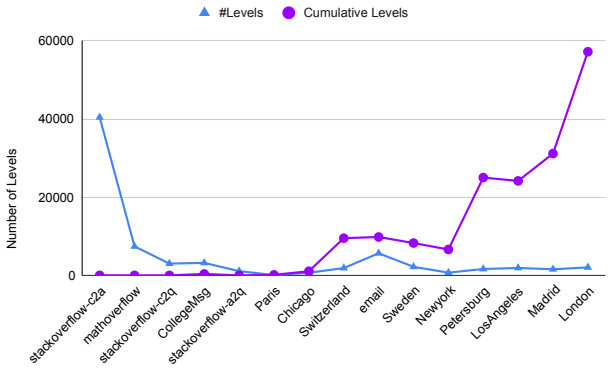

Figure 9: Parameters - No. of levels and cumulative levels

MBFS-Algorithm vs Level order Algorithms.

From Table 5, we observe that LO algorithm performs better than MBFS algorithm on many datasets. To investigate further, we can use $cumulativeLevels$ and $levels$ of a given ESDG to decide which algorithm is more suitable (LO or MBFS).

Whenever $levels < cumulativeLevels$, LO algorithm is a better choice, and MBFS algorithm otherwise. This is evident from Figure 8 and Figure 9. As shown in Figure 9, from the dataset paris onwards, $levels < cumulativeLevels$. We can see in Figure 8 that the running time of LO on these datasets is less than that of MBFS.

6 RELATED WORK

Temporal graphs are very natural and most phenomena can be studied using temporal graphs like the nervous system, power grids, social media networks, and transport networks. There are many types of temporal graphs to model Person-to-person communication, one-to-many information dissemination, physical proximity, cell biology, distributed computing, infrastructural networks, ecological networks, etc [9].

Wu et al., have designed serial algorithms for four minimum path problems in temporal graphs namely *foremost path*, *reverse-foremost path*, *fastest path* and *shortest path* [21]. Many graph transformations [7, 21, 23] are developed to solve single-source shortest and fastest path problems. In all these transformations, a temporal graph is transformed into an equivalent time-respected graph, in which the departure and arrival times of temporal edges are treated as vertices and added edges to capture the necessary dependencies. The key idea in these works is to reduce the number of vertices and edges in the transformed graphs. In this work, we have compared our results with edge-stream, TRG, and ESDG algorithms, which are state-of-the-art techniques to solve FPD problems.

Ni et al., [18] address the earliest-arrival path (EAT) problem with a novel parallel algorithm and a unique data structure known as the Edge Scan Dependency Graph (ESDG). Later, Srikanth et al., have used ESD-graph effectively to solve fastest path problem in the serial setting [19]. We further took inspiration from these results and developed Level Order ESD-graph data structure.

Haryan et al., have designed a GPU-based parallel algorithm to solve earliest arrival time problem [8] using a hierarchical data structure, which is applicable primarily for public transport networks and not for general temporal graphs. Later, Maurya et al., proposed another GPU based solution to solve earliest arrival time problem [17].

Wu et al., have designed a distributed algorithm to solve the earliest arrival path problem using PREGEL model [15]). Due to architectural differences between distributed system and GPU system, these algorithms are not directly applicable to GPU. The design of new algorithms for the fastest path problem in this model is an interesting direction.

Shawi et al., have focused on finding the quickest path in a transportation network [5]. In their work, they assume that each edge has an individual speed and is active all the time, such kind of temporal graphs are used to model private transformation networks. The temporal graphs used in this work assume that an edge is active at specific departure times rather than at all times.

A variety of path problems are being explored in temporal graphs due to their wide applicability. Recently, Jin et al., explored multi-constrained one-to-one path problem on temporal graphs [11].

7 CONCLUSION

Temporal graphs are most suitable for many problems, so we need to treat them differently from static graphs. The fastest path problem is a very important problem in temporal graphs which has multiple use cases in the real world as well as other graph problems. At present, there exist a few serial algorithms to solve it, as well as one multi-core algorithm, but the multi-core algorithm is not suitable for GPU. We introduce three novel GPU-based parallel algorithms for FPD, namely Level Order (LO), Multiple Breadth First Search (MBFS), and Local Work-lists (LW) and implement them on a (GPU) architecture machine. Our GPU algorithms achieve the average speedup of **165**× and the maximum speedup of **1383**× over the state-of-the-art algorithms. We experimentally observe that LO algorithm outperforms MBFS algorithm if and only if the number of levels in an ESD-graph is at most the number of cumulative levels.

REFERENCES

[1] Sara El Alaoui and Byrav Ramamurthy. 2017. EAODR: A novel routing algorithm based on the Modified Temporal Graph network model for DTN-based Interplanetary Networks. *Comput. Networks* 129 (2017), 129–141. https://doi.org/10.1016/J.COMNET.2017.09.012

[2] Simon Bourigault, Cédric Lagnier, Sylvain Lamprier, Ludovic Denoyer, and Patrick Gallinari. 2014. Learning social network embeddings for predicting information diffusion. In *Seventh ACM International Conference on Web Search and Data Mining, WSDM 2014, New York, NY, USA, February 24-28, 2014*, Ben Carterette, Fernando Diaz, Carlos Castillo, and Donald Metzler (Eds.). ACM, NY, USA, 393–402. https://doi.org/10.1145/2556195.2556216

[3] Rui Dai, Shenkun Xu, Qian Gu, Chenguang Ji, and Kaikui Liu. 2020. Hybrid Spatio-Temporal Graph Convolutional Network: Improving Traffic Prediction with Navigation Data. In *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, Rajesh Gupta, Yan Liu, Jiliang Tang, and B. Aditya Prakash (Eds.). ACM, CA, USA, 3074–3082. https://doi.org/10.1145/3394486.3403358

[4] Daniel Delling, Bastian Katz, and Thomas Pajor. 2012. Parallel computation of best connections in public transportation networks. *ACM J. Exp. Algorithmics* 17, Article 4.4 (oct 2012), 26 pages. https://doi.org/10.1145/2133803.2345678

[5] Radwa El Shawi, Joachim Gudmundsson, and Christos Levcopoulos. 2014. Quickest path queries on transportation network. *Computational Geometry* 47, 7 (2014), 695–709.

[6] Sanaz Gheibi, Tania Banerjee, Sanjay Ranka, and Sartaj Sahni. 2021. An Effective Data Structure for Contact Sequence Temporal Graphs. In *2021 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, Athens, Greece, 1–8. https://doi.org/10.1109/ISCC53001.2021.9631469

[7] Sanaz Gheibi, Tania Banerjee, Sanjay Ranka, and Sartaj Sahni. 2024. Path Algorithms for Contact Sequence Temporal Graphs. *Algorithms* 17, 4 (2024), 1–19. https://doi.org/10.3390/a17040148

[8] Chirayu Anant Haryan, G. Ramakrishna, Rupesh Nasre, and Allam Dinesh Reddy. 2020. A GPU Algorithm for Earliest Arrival Time Problem in Public Transport Networks. In *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, Virtual, 171–180. https://doi.org/10.1109/HiPC50609.2020.00031

[9] Petter Holme and Jari Saramäki. 2012. Temporal networks. *Physics reports* 519, 3 (2012), 97–125.

[10] Yushan Jiang, Shuteng Niu, Kai Zhang, Bowen Chen, Chengtao Xu, Dahai Liu, and Houbing Song. 2022. Spatial–Temporal Graph Data Mining for IoT-Enabled Air Mobility Prediction. *IEEE Internet of Things Journal* 9, 12 (2022), 9232–9240. https://doi.org/10.1109/JIOT.2021.3090265

[11] Yue Jin, Zijun Chen, and Wenyuan Liu. 2024. Enumerating all multi-constrained s-t paths on temporal graph. *Knowl. Inf. Syst.* 66, 2 (2024), 1135–1165. https://doi.org/10.1007/S10115-023-01958-8

[12] Hyoungshick Kim and Ross Anderson. 2012. Temporal node centrality in complex networks. *Physical Review E* 85, 2 (2012), 026107.

[13] Youru Li, Zhenfeng Zhu, Xiaobo Guo, Linxun Chen, Zhouyin Wang, Yinmeng Wang, Bing Han, and Yao Zhao. 2023. Learning Joint Relational Co-evolution in Spatial-Temporal Knowledge Graph for SMEs Supply Chain Prediction. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2023, Long Beach, CA, USA, August 6-10, 2023*, Ambuj K. Singh, Yizhou Sun, Leman Akoglu, Dimitrios Gunopulos, Xifeng Yan, Ravi Kumar, Fatma Ozcan, and Jieping Ye (Eds.). ACM, CA, USA, 4426–4436. https://doi.org/10.1145/3580305.3599855

[14] Xiaoxiao Ma, Jia Wu, Shan Xue, Jian Yang, Chuan Zhou, Quan Z Sheng, Hui Xiong, and Leman Akoglu. 2021. A comprehensive survey on graph anomaly detection with deep learning. *IEEE Transactions on Knowledge and Data Engineering* 35, 12 (2021), 12012–12038.

[15] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, Indianapolis, Indiana, USA, 135–146.

[16] Junbin Mao, Hanhe Lin, Xu Tian, Yi Pan, and Jin Liu. 2023. FedGST: Federated Graph Spatio-Temporal Framework for Brain Functional Disease Prediction. In *2023 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE, Turkey, 1356–1361. https://doi.org/10.1109/BIBM58861.2023.10385983

[17] Sunil Kumar Maurya and Anshu S Anand. 2022. A Novel GPU-Based Approach to Exploit Time-Respectingness in Public Transport Networks for Efficient Computation of Earliest Arrival Time. *IEEE Access* 10 (2022), 81877–81887.

[18] Peng Ni, Masatoshi Hanai, Wen Jun Tan, Chen Wang, and Wentong Cai. 2017. Parallel algorithm for single-source earliest-arrival problem in temporal graphs. In *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, IEEE, Bristol, United Kingdom, 493–502.

[19] Mithinti Srikanth and G. Ramakrishna. 2024. Efficient Algorithms for Earliest and Fastest Paths in Public Transport Networks. arXiv:2404.19422

[20] Haoran Wang, Licheng Jiao, Fang Liu, Lingling Li, Xu Liu, Deyi Ji, and Weihao Gan. 2023. Learning Social Spatio-Temporal Relation Graph in the Wild and a Video Benchmark. *IEEE Trans. Neural Networks Learn. Syst.* 34, 6 (2023), 2951–2964. https://doi.org/10.1109/TNNLS.2021.3110682

[21] Huanhuan Wu, James Cheng, Yiping Ke, Silu Huang, Yuzhen Huang, and Hejun Wu. 2016. Efficient Algorithms for Temporal Path Computation. *IEEE Transactions on Knowledge and Data Engineering* 28, 11 (2016), 2927–2942. https://doi.org/10.1109/TKDE.2016.2594065

[22] Qiang Wu, Yuanze Du, Hua Xu, Yingwang Zhao, Xiaoyan Zhang, and Yi Yao. 2020. Finding the earliest arrival path through a time-varying network for evacuation planning of mine water inrush. *Safety Science* 130 (2020), 104836.

[23] Philipp Zschoche, Till Fluschnik, Hendrik Molter, and Rolf Niedermeier. 2020. The complexity of finding small separators in temporal graphs. *J. Comput. System Sci.* 107 (2020), 72–92.