Received 12 June 2022, accepted 10 July 2022, date of publication 19 July 2022, date of current version 9 August 2022.
Digital Object Identifier 10.1109/ACCESS.2022.3192443

APPLIED RESEARCH

A Novel GPU-Based Approach to Exploit Time-Respectingness in Public Transport Networks for Efficient Computation of Earliest Arrival Time

SUNIL KUMAR MAURYA AND ANSHU S. ANAND, (Senior Member, IEEE)
Department of Information Technology, IIIT Allahabad, Prayagraj 211015, India
Corresponding author: Anshu S. Anand (anshu@iiita.ac.in)

ABSTRACT In static temporal networks, the Earliest Arrival Time (EAT) problem is to calculate the earliest possible time of arrival at a set of vertices from a given source vertex. Applications of the EAT problem include designing efficient evacuation planning in dynamic scenarios, optimal journey planning in transport networks, and optimal flow management in supply chains. There exist several solutions for the EAT problem in the literature, however, there is limited work on GPU (Graphics Processing Unit) based solutions to leverage the capabilities of the high throughput accelerator for better performance. Further, there is also a need for more efficient methods to process the inherent earliest arrival dependencies in a transport network. In this paper, we propose a suite of five incremental (GPU) algorithms for the one-to-all Earliest Arrival Time problem in public transport networks. The Selective-check-version is the most optimized approach and hence, the key algorithm. It uses an edge coloring based approach to trace the time-respectingness of paths and processing the in-edges in a sorted order based on their arrival times. Its key characteristic is that it is very fast for the best-case networks where all temporal paths are *time-respecting*. For the Selective-check version, we observed an average speedup of 6.45 against the Serial Connection-scan algorithm and 2.77 w.r.t. the Edge-version algorithm.

INDEX TERMS Earliest arrival time, GPU, shared memory, temporal graph, transport network.

I. INTRODUCTION

Seeking optimal routes is perhaps the most common and daunting task in any network. The optimal routes could be the shortest, fastest, earliest arrival, etc. Designing a serial algorithm is genuinely a good approach to solve it, but it is not much feasible for large networks. The Earliest Arrival Time (EAT) problem is a category of route-planning problems associated with finding the least possible timestamps at which a set of destination vertices is reached after departing from a source vertex. For a transport network G, a source vertex s, and a scheduled timetable for each of the vehicles, the EAT problem is to plan the routes in such a way that we can reach the destination vertex at the earliest. This may require one to change several vehicles along the route. In general, a transport network is comprised of several types of vehicles but there can also be networks consisting of a single vehicle or identical vehicles of the same type, e.g., a cargo shipping network comprises vehicles of identical types.

The associate editor coordinating the review of this manuscript and approving it for publication was Sun-Yuan Hsieh.

Though the EAT problem is different from Single Source Shortest Path (SSSP) Problem, but still the works for SSSP problem proves significant here as evidenced in [11], [12], [14], [16], and [17]. Besides, a good graph model and the graph process techniques matter a lot in having an efficient problem solving mechanism [18]–[20]. The graph with several attributes may need deeper analysis to fully understand it, hence, the centrality measures and multi-layer analysis may prove useful [13], [15].

In addition to these, the EAT problem has various applications. For example,

- The EAT problem has significance in optimal journey planning for the single source to multiple destinations EAT problem. Its solution requires a specialized variant of the one-to-all EAT problem's solution.
- EAT problem is useful in solving cost-effective travel problems under various dynamics like weather conditions, peak hours, holidays, route diversions etc.
- Another application is evacuation planning where a set of the earliest reachable spots can be identified as an exit spot.
- Designing optimal flow management in supply chains. It may include planning a set of paths for the earliest transportation of a specific volume of goods.

A naive way to solve the EAT problem is to scan each temporal edge sequentially in a breadth-first search fashion and update the earliest arrival time of neighboring vertices. Inspired by it, Dibbelt *et al.* proposed the Connection-Scan algorithm [1] to calculate the minimum expected arrival time. The variants of Dijkstra's algorithm can also be useful in calculating EAT [2]. These are serial algorithms, easy-to-use but suffering from no parallelism. The one-to-all profile search problem can be parallelized as follows:- if each vertex $v \in V$ is scanned sequentially in a breadth-first search fashion but all the outgoing edges of v are scanned in parallel to update the earliest arrival value of out-neighbors of v [4].

The difficulty that rise while parallelizing the transport network is the earliest arrival dependency of a vertex on another vertex. In other words, the earliest arrival time of a vertex v can be dependent on the earliest arrival time of one of the in-neighbors of v. To deal with it efficiently, Ni *et al.* proposed the ESDG algorithm [6]. To deal with the problem of limited parallelism in contemporary works, Ramakrishna *et al.* proposed a set of incremental algorithms to solve the EAT problem [5].

Our contributions in this paper is summarized as follows:-

- We propose a set of incrementally designed GPU algorithms for the EAT problem in a time-table based transport networks.
- We present a new EAT problem-specific temporal graph representation for better performance. It helps in visualizing and representing each vertex and its in-edges as an independent entity.
- We experimentally evaluate the existing as well as the proposed algorithm on real-world datasets and compare their execution times, speedups, and the bytes of the shared memory expended.

The rest of the paper is organized as follows. In Section II, we introduce the basic concepts of temporal graphs and some of the existing methods to solve EAT problem. In Section III, we propose the incremental algorithms to solve the earliest arrival problem in temporal graphs. In Section IV, we show the experimental evaluation of the proposed algorithms and existing algorithms. Conclusion and future work are presented in Section V.

II. PRELIMINARIES

We first formally define a temporal graph and describe its similarity with transport networks. Later, we introduce the baseline algorithms for EAT problem, namely, Connection-scan algorithm [1], Parallel SPCS (Self Pruning Connection Setting) algorithm [4], RAPTOR (Round bAsed Public Transit Optimized Router) algorithm [3], Parallel Connection-Scan algorithm [5], ESDG (Edge-Scan Dependency Graph) algorithm [6], and Edge-version algorithm [5].

A temporal network differs from a traditional network structures where the latter networks only keep the information of the vertices and the physical/logical edges connecting them. On the other hand, a temporal network represents the flow of objects among the set of vertices it has. It is an abstract network structure representing the flow/movement of objects happening in the traditional network. A temporal graph G is represented by a set of vertices V and temporal edges E respectively, where each edge model a 4-tuple information. Each tuple (u, v, t, d) corresponding to E represents:-

u = source vertex
v = destination vertex
t = departure time
d = duration time

Semantically, each tuple (u, v, t, d) represents a vehicle going from u to v and that departs from u at time t and the average duration it takes to reach v is equal to d.

A transport network can be represented as a temporal graph where the set of stops can be represented as the set of vertices, while the movement of the vehicle at time t from stop u to stop v, when it takes d duration to reach v can be represented by a temporal edge from vertex u to v labeled with a tuple (t, d). This information can also be stored in 4-tuple form as (u, v, t, d), where u, v, t, d has their respective meaning same as that of temporal graphs.

The state-of-the-art algorithms for EAT problems include Serial Connection-scan algorithm [1], Parallel SPCS algorithm [4], RAPTOR algorithm [3], Parallel Connection-Scan algorithm [5], ESDG algorithm [6], and Edge-version algorithm [5], which we described below.

A. SERIAL CONNECTION-SCAN ALGORITHM

The Connection-scan algorithm [1] is the first recognized work in the context of EAT. As per this algorithm, assume that the earliest arrival value of s is the same as its departure time, t_s and the EAT at each vertex is ∞. Each edge is sorted in ascending order of departure times. Further, each edge is scanned sequentially starting from the edge with the lowest departure time to update the earliest arrival value of neighboring vertices [1].

This algorithm is easy to understand and known for its simplicity but it has no parallelism which makes it less useful on large networks.

B. PARALLEL SPCS ALGORITHM

This parallel algorithm was designed for the profile-search earliest arrival time problem [4]. The core approach is to keep track of the set of outgoing connections for each vertex

$v \in V$. Let this set be represented by $conn(v)$. The algorithm further proceeds by partitioning the $conn(v)$ into k subsets, each of these subsets is processed by an independent thread. The partial results computed are afterward combined to get accurate results.

The algorithm is good for static networks only. The idea of parallelism is kept limited to few multi-core CPU setup.

C. RAPTOR ALGORITHM

The RAPTOR algorithm is a bi-criteria minimization algorithm that aims to calculate the earliest arrival time with the least possible transfers. It works in rounds and in the k^{th} round, the fastest way of reaching every vertex with at most $k - 1$ transfers is computed [3]. The algorithm is great to be applied to dynamic scenarios and in multi-criteria profile search problems, but it lacks parallelism.

D. PARALLEL CONNECTION-SCAN ALGORITHM

This algorithm is very similar to the serial connection-scan algorithm except for the fact it is a naive parallelization of CSA. Here, each of the edges is scanned in parallel. Parallel scanning is done multiple times until there is no change is observed in the earliest arrival times of any vertex in consecutive iterations [5]. Its biggest disadvantage is that it highly depends on the no. of edges. More the edges, more the number of threads is required.

E. ESDG ALGORITHM

It is a graph transformation based algorithm [6] which indirectly addresses the inherent dependencies among the temporal edges. It transforms an original graph G into an *edge-scan-dependency-graph* G'. Later, for every level in G', each thread is assigned. Every thread i computes the earliest arrival time of all vertices belonging to i^{th} level.

The ESDG algorithm is specifically designed for the multi-core systems, not as large as that of GPU cores, hence, it exhibits low parallelism. In addition, the graph transformation process adds extra overhead as well.

F. EDGE-VERSION ALGORITHM

This algorithm was one of the most efficient algorithms among the incremental algorithms proposed by Ramakrishna *et al.* In this approach, all the edges are preprocessed into connection-types, clusters, and arithmetic progressions [5]. With this technique, looking up for the edge with the minimum arrival time output is improved. Further, each of the vertex pairs (u, v) directly connected by an edge is processed in parallel. All the edges between (u, v) are processed by one thread. Each thread will look for the edge with the minimum arrival time output for their respective (u, v) pairs to relax the arrival time of v. This parallel processing is iterated multiple times until no change is observed in consecutive iterations.

The algorithm is great in terms of parallelism, pruning techniques, and speedup but it is applicable only on static networks. For its applicability to dynamic networks, the algorithm should be rerun every time the network changes.

Across all these baseline algorithms, we noticed that there is the need of a very fast method to process the earliest arrival dependencies between any two vertices. Besides, there is limited work on GPU-based solutions for the EAT problem. Most of the current approaches are based on multi-core CPU setups that show limited parallelism and limited speed-ups. It is also observed that the EAT problem in networks closer to the best case can be solved in $O(1)$ time, but it is absent in existing works.

III. INCREMENTAL PARALLEL ALGORITHM

In this section, we describe our incrementally designed parallel algorithm for calculating the earliest arrival time from a source vertex to all the vertices. Firstly, we will represent the transport network as $G = (V, E)$, where, V = Set of vertices E = Set of temporal edges Each temporal edge in set E can be represented as a set of 4-tuples (u, v, t, d), where,

u = The vertex endpoint from where the vehicle departs
v = The vertex endpoint the vehicle is destined to
t = departure time of a vehicle
d = duration time from u to v

We proposed our main algorithm through 5 incremental variants. These are:-

- Vertex-version
- Time-scale version
- Multi-Time-scale version
- Parallel-backward-check-version
- Selective-check version

PROPOSED GRAPH REPRESENTATION

For better performance, we propose a new representation technique to store and process the temporal graphs for transport networks.

In the proposed algorithms (except Vertex-version), we use a dedicated array S_v for all the vertices $v \in V$ to store information of its in-neighbors, departure time, and arrival time at v from its in-neighbors. All the in-neighbors u of v and the departure time and arrival time of $u \rightarrow v$ edge is stored in the form of a 3-tuple, as shown in Figure 1b.

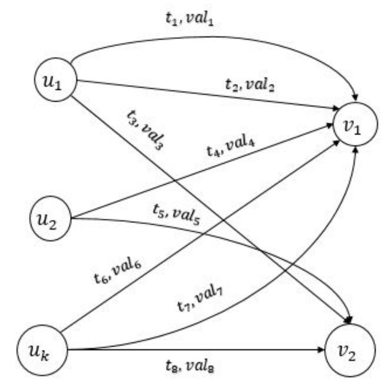
Consider the subgraph as shown in Figure 1a. For this subgraph, Figure 1b illustrates the representation for vertices v_1 and v_2.

Such a representation helps in visualizing each vertex and the associated set of data as an independent unit which ultimately eases the task of computing the earliest arrival times.

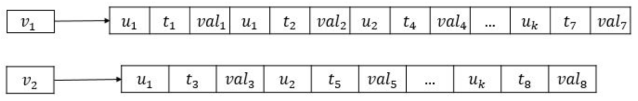
A. VERTEX-VERSION

In this version, each vertex is processed in parallel, and for all of these vertices, each outgoing edge is processed in parallel to update the EAT of its out-neighbors. Thus, for two vertices u and v such that $u \in v_{out}$, if $e[v] \leq t_{v,u}$ and $e[u] > t_{v,u} + dur_{v,u}$ then $e[u]$ is set to $t_{v,u} + dur_{v,u}$, where

v_{out} = Set of outgoing vertices of v



(a) An example graph.


v_1
u_1 t_1 val_1 u_1 t_2 val_2 u_2 t_4 val_4 ... u_k t_7 val_7
v_2
u_1 t_3 val_3 u_2 t_5 val_5 ... u_k t_8 val_8

(b) For the vertex v, its in-neighbours and departure and arrival times of the in-edges are stored in this way.

FIGURE 1. Illustration of technique to store the graph.

$e[v]$ = Earliest arrival time of vertex v
$t_{v,u}$ = Departure time of (v, u) edge
$dur_{v,u}$ = Duration of (v, u) edge

Algorithm 1 Vertex-Version

```
Input: $G = (V, E)$.
Output: Earliest Arrival Times of all vertices in $G$.
1: for all $v \in V$ do                                  ▷ do in parallel
2:     for all $u \in v_{out}$ do                         ▷ do in parallel
3:         for all edges $(v, u)$ do
4:             if $e[v] \leq t_{v,u}$ and $e[u] > t_{v,u} + dur_{v,u}$ then
5:                 $e[u] = t_{v,u} + dur_{v,u}$
6: Go to step 1 if at least one $e[v]$ is changed.
```

Complexity:- The steps in Algorithm 1 take $O(1)$ time because it is done in parallel but the serial iterations required in Algorithm 1 may go to $|V|$ iterations in the worst case, where $|V|$ = no. of vertices. Hence, its time complexity is $O(|V|)$.

B. TIME-SCALE VERSION

In this algorithm, we use a time scale for the processing of vertices. The heuristic is to make a time scale T of length 1440, where the i^{th} index stores the vertices that are reached at the i^{th} minute. We assume that the schedule in the given network repeats itself every 24 hours. Since the total no. of minutes in a 24-hour clock $= 24 \times 60 = 1440$, that's why the length of T is set to 1440.

The algorithm proceeds with the initialization of a time scale T followed by the for loops to populate T by processing the entire graph. In steps 2-7, we scan all the vertices $v \in V$ in parallel in the given graph and for each of that v, its in-neighbors are picked in parallel to scan all the incoming edges to v. Note that, v_{in} denotes the set of in-neighbors of v. For each of the in-edges of a vertex v, the minimum arrival time to v from any of its in-edge is noted down, say it AT. After that, at index AT in T, this vertex v is stored. After T is filled, each of its rows is scanned in parallel to process the vertices stored there. The i^{th} thread will process all the vertices stored at the i^{th} index in T and hence, tries to minimize its EAT to i. The lock-unlock mechanism is used additionally to ensure the atomicity.

Note that, it is designed under the assumption that all the paths in the given network are *time-respecting*. Hence, it is not useful in real-life transport networks. Since its theory is useful in designing the subsequent algorithms, therefore, it is necessarily considered here.

Algorithm 2 Time-Scale Version

```
Input: $G = (V, E)$.
Output: Earliest Arrival Times of all vertices in $G$.
 1: Initialize a Time scale $T$ of rows 1440.
 2: for all $v \in V$ do                                  ▷ do in parallel
 3:     for all $u \in v_{in}$ do                          ▷ do in parallel
 4:         AT = $\infty$
 5:         for all edges $(u, v)$ do
 6:             AT = min(AT, $t_{u,v} + dur_{u,v}$)
 7:         Add the vertex $v$ in $T$[AT]
 8: for all $i \in [0, 1440)$ do                          ▷ do in parallel
 9:     for all $v$ stored at $T$[i] do
10:         if $e[v] > i$ then
11:             LOCK($v$)
12:             $e[v] = i$
13:             UNLOCK($v$)
```

Complexity:- The steps 2 to 7 in Algorithm 2 take $O(1)$ time because it is done in parallel while computation ahead of the step 7 may take $O(|V|)$ time, where $|V|$ = no. of vertices. The reason behind the $O(|V|)$ time is the LOCK-UNLOCK mechanism that's forcing some threads to have sequential ordering. In the worst case, this sequential ordering may turn out to be $O(|V|)$ length.

C. MULTI-TIME-SCALE VERSION

In this version, we improved the idea of the time scale few more steps. Here, each vertex is treated as an independent unit whose EAT entirely depends on its in-neighbors. Each vertex is provided with its own time scale to store its in-neighbors, departure, and arrival times of the corresponding in-edge.

The algorithm proceeds with the parallel scanning of every vertex allocating each of them a time scale of length 1440. Further, all the in-edges of these vertices are scanned to find the in-edge with the minimum arrival time, represented by idx. After that, all the in-neighbors corresponding to the in-edges with idx arrival time is pushed into the list corresponding to T[idx]. The push operation into T[idx] is represented by *push*. Note that $t_{(X,Y)}$ and $dur_{(X,Y)}$ represent the departure time and duration of an edge from X to Y respectively. To have an atomic read-write, T[idx] is locked before adding any vertex into it. After the addition to it, T[*idx*] is unlocked.

Further, for all the vertex $X \in V$, its time scale is scanned sequentially. At the very first index (*idx*) in its time scale where there exists at least one vertex, say *depNode*, is noted. Later, the vertex *depNode* is checked whether it has achieved its earliest arrival time or not. This condition is checked with the use of the *verified* array. The i^{th} index of *verified* is *true* when the i^{th} vertex has achieved its EAT, otherwise, it is *false*. When *verified*[*depNode*] is *true* and the earliest arrival time at *depNode* is less than or equal to the departure time of the corresponding edge from *depNode* to X, the EAT of X is set to *idx* and *verified*[X] is set to *true* followed by the stopping of time-scale scanning. In another case, if *verified*[*depNode*] is still *false*, the vertex X waits for the *depNode* for its EAT calculation.

Algorithm 3 Multi-Time-Scale Version

```
Input: $G = (V, E)$.
Output: Earliest Arrival Times of all vertices in $G$.
 1: for all $X \in V$ do                                  ▷ do in parallel
 2:     T[1440] = $\infty$
 3:     idx = $\infty$
 4:     for all $Y \in V$ do                              ▷ do in parallel
 5:         for all edges $(Y, X)$ do
 6:             idx = min(idx, $t_{(Y,X)} + dur_{(Y,X)}$)
 7:         LOCK(T[idx])
 8:         T[idx].push(Y, $t_{(Y,X)}$)
 9:         UNLOCK(T[idx])
10: for all $X \in V$ do                                  ▷ do in parallel
11:     for all $idx \in [0, 1440)$ do
12:         for all $depNode \in T[idx]$ do
13:             if $depNode \neq \infty$ then
14:                 if verified[depNode] = TRUE then
15:                     if e[depNode] $\leq t_{(depNode,X)}$ then
16:                         $e[X] = idx$
17:                         verified[X] = TRUE
18:                         break time-scale scanning
19:                 else
20:                     wait on depNode
```

Complexity:- In Algorithm 3, the time complexity highly depends on the waiting of a vertex for its in-neighbors. Such waiting contributes to the creation of a waiting chain. This waiting chain is as long as the diameter of the graph. Thus, its complexity is $O(D)$ where D = The diameter of the graph.

D. PARALLEL BACKWARD-CHECK VERSION

Though Algorithm 3 provides faster access to EATs, it results in wastage of space. Thus, to optimize space usage, we introduce Algorithm 4. In this version, we optimize the part of time-scale in Algorithm 3. Rather than having a time scale of fixed length for every vertex, a list can be used for every vertex to store in-neighbors and arrival time in the form of key-value pairs. For faster access, shared memories can be used instead of such a list.

The algorithm proceeds by the scanning of every vertex in parallel followed by the parallel scanning of all the in-edges corresponding to every vertex. The in-neighbors and the arrival times of the in-edges are stored in the shared memory dedicated for every vertex, represented by $shmdmem_v$. Note that the respective in-neighbors are stored in the form of 'keys' while the respective arrival time is stored in the form of 'values'. Next, each vertex is scanned in parallel and $\forall v \in V$, the stored *key* and *value* in the corresponding shared memories are extracted. Later this *key* vertex is checked whether the *verified*[*key*] is *true* or not. The concept of *verified* is the same as it was explained in Algorithm 3. If the EAT at v, represented by $e[v]$, is less than or equal to the departure time of the edge (key, v), then the value of $e[v]$ is set with *value* if it is less than the previous value of $e[v]$, otherwise, there is no need to change the value of $e[v]$. If *verified*[*key*] is not *true* when it is checked by the thread corresponding to vertex v then, it waits for the *verified*[*key*] until it is set to *true* by its respective thread. When it is set to *true*, the above process is repeated again.

Algorithm 4 Parallel Backward-Check-Version

```
Input: $G = (V, E)$.
Output: Earliest Arrival Times of all vertices in $G$.
 1: for all $v \in V$ do                                  ▷ do in parallel
 2:     for all $u \in v_{in}$ do                         ▷ do in parallel
 3:         shmdmem$_v$.add($u$, $t_{u,v} + dur_{u,v}$)
 4: for all $v \in V$ do                                  ▷ do in parallel
 5:     flag = FALSE
 6:     for all $(key, val) \in shmdmem$ do               ▷ do in parallel
 7:         if verified[key] = TRUE then
 8:             if $e[key] \leq t_{key,v}$ then
 9:                 $e[v] = min(e[v], val)$
10:                 flag = TRUE
11:         else
12:             WAIT on key until its respective thread sets
    the verified[key]
13:             Go to step 3 when verified[key] = TRUE
14:     verified[v] = (verified[v] OR flag)
```

Complexity:- For similar reasons as in the Algorithm 3, the time complexity of Algorithm 4 will be $O(D)$ time for the real-life transport networks

E. SELECTIVE-CHECK VERSION

The Selective-check algorithm relies on our key observation that in the calculation of the earliest arrival time for a vertex v, it is not always needed to calculate the earliest arrival time at all vertices u, where $u \in V_{in}$ and V_{in} is the set of all in-neighbors of v. We illustrate this with an example. Consider an edge, $u \rightarrow v$ that gives the minimum arrival time at v, and there exists some *time-respecting* path [6], $s \rightarrow w_1 \rightarrow w_2 \rightarrow \cdots \rightarrow u$, not necessarily the earliest arrival path, then it can be inferred that the path $s \rightarrow w_1 \rightarrow w_2 \rightarrow \cdots \rightarrow u \rightarrow v$ is the earliest arrival path to v.

Algorithm 5 Selective-Check Version

Input: $G = (V, E)$.
Output: Earliest Arrival Times of all vertices in G.

```
1: for all $v \in V$ do                                  ▷ do in parallel
2:     for all $u \in v_{in}$ do                          ▷ do in parallel
3:         $arr_v.add(u, dt_{(u,v)}, at_{(u,v)})$
4: for all $v \in V$ do
5:     Sort $arr_v$ in ascending order of arrival times
6: for all $v \in V$ do                    ▷ Scan each vertex $v$ in parallel
7:     found = false
8:     for all $i \in [0, arr_v.size())$ do ▷ Scan in-neighbor of $v$
9:         $(u, dt, at) = (arr_v[i]_0, arr_v[i]_1, arr_v[i]_2)$
10:        for all $j \in [0, arr_u.size())$ do
11:            $(w, dt', at') = (arr_u[j]_0, arr_u[j]_1, arr_u[j]_2)$
12:            if $color[w, u, dt', at'] = white$ then
13:                wait for $(w, u, dt', at')$
14:            if $dt \geq at'$ AND $color[w, u, dt', at'] = green$ then
15:                $e[v] = at$
16:                found = true
17:                $color[u, v, dt, at] = green$
18:                break
19:        if found then
20:            break
21:        else
22:            $color[u, v, dt, at] = red$
```

The algorithm starts by processing all the edges in parallel to store all the arrival times at vertex v into an array dedicated for v, denoted by arr_v, $\forall v \in V$ (Lines 1-3). Note that u denotes the in-neighbor of v while $dt_{(u,v)}$ and $at_{(u,v)}$ denotes the departure time and the arrival time of an edge from u to v. Next, the arrays for each vertex v, arr_v are sorted in parallel in ascending order of arrival times (Lines 4-5). Following this, each vertex is processed in parallel to compute the EATs. For every vertex $v \in V$, its stored in-neighbors u and the corresponding departure and arrival times of the in-edge $u \rightarrow v$ is extracted. This operation is done in Line 9 in tuple form as shown below:-

$$(u, dt, at) = (arr_v[i]_0, arr_v[i]_1, arr_v[i]_2)$$

Here, u, dt and at receive the values from $arr_v[i]_0$, $arr_v[i]_1$ and $arr_v[i]_2$ respectively. Note that $arr_v[i]_0$ denotes the i^{th} in-neighbour of v. Similarly, $arr_v[i]_1$ and $arr_v[i]_2$ denote the departure time and the arrival time of an in-edge from the i^{th} in-neighbor to v respectively.

The $color[]$ array is used to keep track of the utility of a temporal edge (u, v, t, d), where each symbol has the same meaning as described in Section II. Note down that $color[u, v, t, d]$ can be $white$, red, or $green$ depending on whether the edge (u, v, t, d) is in the $unprocessed$, $not-usable$, or $processed$ states respectively. An edge will be in the $unprocessed$ state if it is not determined yet whether s to v path via (u, v, t, d) is *time-respecting* or not. It will be in the $not - usable$ state if this edge is found to be not *time-respecting*, i.e, there does not exist any path from s to u with an arrival time less than or equal to t, the departure time from u. Further, an edge will be in the *processed* state if there exists at least one *time-respecting* path from s to u with respect to the edge from u to v.

The algorithm scans each vertex in parallel (Line 6) and $\forall v \in V$, its in-neighbor u, and the departure time and arrival time of the corresponding u to v edge is extracted from the arr_v sequentially (Line 9). Next, each of the in-neighbors and the corresponding in-edge to u is scanned in arr_u (Lines 10-11). The state of each of the temporal in-edges, $w \rightarrow u$ of u is then checked and if it is $white$, the vertex v needs to wait for the state to change to $red/green$ (Lines 12-13). On the other hand, if the departure time of edge $u \rightarrow v$ is greater than or equal to the arrival time of edge $w \rightarrow u$, and the state of the edge $w \rightarrow u$ is $green$, then the EAT of v is set to the arrival time of the edge $u \rightarrow v$ (Line 15). Since the edge $u \rightarrow v$ leads to a time-respecting path from s to v, $color[u, v, dt, at]$ is set to $green$ and $found$ is set to $true$ (Lines 16-17) to indicate that the earliest arrival path to v is found.

Subsequently, the further scanning of other in-edges of v is immediately stopped because the earliest arrival path is achieved and hence, further scanning is irrelevant. It may also happen that for the edge $u \rightarrow v$, there doesn't exist any time-respecting paths from s to u. In such a case, $color[u, v, dt, at]$ is set to red (Line 22) indicating that v is not reachable from s through the edge $u \rightarrow v$.

Complexity:- The inherent waiting chain in this algorithm make its time complexity to reach $O(D)$, where D = The diameter of the graph.

1) ILLUSTRATIVE EXAMPLE

Consider a sample network with the journey times as shown in Fig. 2. The labeled edges represent departure and arrival times respectively. For example, the edge labeled as (10:30 am, 11:00 am) between (S, 2) represents a vehicle departing from station S at 10:30 am and arriving at station 2 at 11:00 am.

The Selective-check algorithm starts by grouping all the in-edges for every vertex. Further, it sorts all those in-edges on arrival times. Sorting ensures that the in-neighbor resulting in the earliest arrival time is accessed first. This is shown in Fig. 3.

As per Algorithm 5, a GPU thread corresponding to a vertex will be dependent on threads processing its in-neighbors to ensure time-respectingness. For example, the thread for vertex 9 will wait for the threads of vertices 7 and 8 to indicate the presence of at least one time-respecting path to them. As evident from Figure 1, for vertex 9, the possible time-respecting paths are $S \rightarrow 1 \rightarrow 4 \rightarrow 7$, $S \rightarrow 2 \rightarrow 4 \rightarrow 7$, $S \rightarrow 3 \rightarrow 5 \rightarrow 7$ and $S \rightarrow 3 \rightarrow 5 \rightarrow 8$. As there exist three in-edges to vertex 9, the edges $6 \rightarrow 9$,

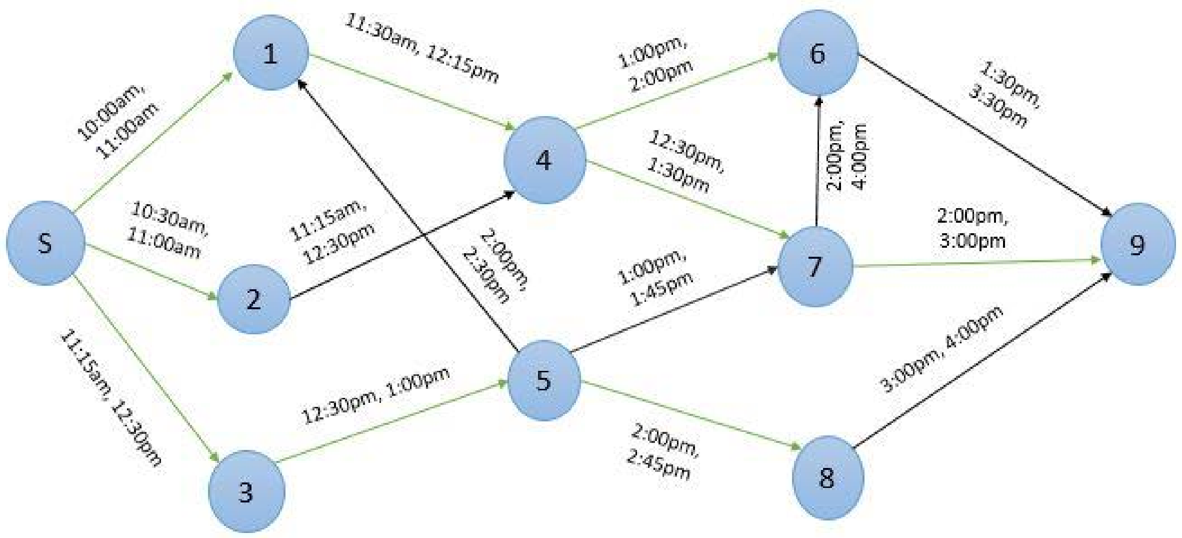
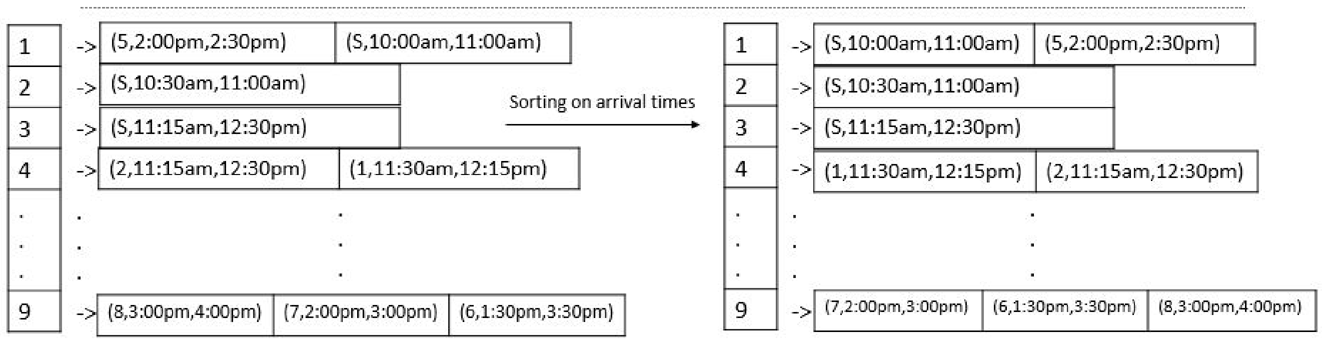


FIGURE 2. An example to illustrate the working of Selective-check algorithm.



1	->	(5,2:00pm,2:30pm)	(S,10:00am,11:00am)	
2	->	(S,10:30am,11:00am)		
3	->	(S,11:15am,12:30pm)		
4	->	(2,11:15am,12:30pm)	(1,11:30am,12:15pm)	
.	.	.		
.	.	.		
.	.	.		
9	->	(8,3:00pm,4:00pm)	(7,2:00pm,3:00pm)	(6,1:30pm,3:30pm)

Sorting on arrival times

1	->	(S,10:00am,11:00am)	(5,2:00pm,2:30pm)	
2	->	(S,10:30am,11:00am)		
3	->	(S,11:15am,12:30pm)		
4	->	(1,11:30am,12:15pm)	(2,11:15am,12:30pm)	
.	.	.		
.	.	.		
.	.	.		
9	->	(7,2:00pm,3:00pm)	(6,1:30pm,3:30pm)	(8,3:00pm,4:00pm)

FIGURE 3. State of Data structure before and after sorting.

$7 \rightarrow 9$, and $8 \rightarrow 9$ are checked for time-respectingness. Since the edge $7 \rightarrow 9$ results in the earliest arrival at vertex 9 assuming time-respectingness, it is processed first (as it was sorted earlier, it appears at the head of the list). As soon as it is determined that there exists a time-respecting path (and therefore a green-colored path) from S to vertex 7, and if the arrival time through this path is at least the same as the departure time from 7, the time-respectingness of the edge from vertex 7 to 9 is established and therefore colored green, and the EAT at vertex 9 updated to 3:00 pm. The other edges from vertices 6 and 8 will not be processed by the algorithm and therefore, remains white (shown as black in the figure).

2) CORRECTNESS

We now establish the proof of correctness for the Selective-check version algorithm. We assume that for any vertex v, there exists a unique EAT path from s to v. Though this need not be true in practice, it is safe to make this assumption since even in the presence of multiple EAT paths between the same pair of vertices, the algorithm will only identify a single path due to the ordering imposed by sorting.

Lemma 1: If $p = (s, v_1, v_2, \ldots, v_m, v)$ is the earliest arrival path from s to v, then $p' = (s, v_1, v_2, \ldots, v_{m-1}, v_m)$ is the earliest arrival path from s to v_m.

Proof: We prove it by contradiction. Suppose p' is not the EAT path from s to v_m, then v_m should have an in-neighbor, v_k, other than v_{m-1} that returns the least arrival time at v_m and it should be an intermediate vertex in some *time-respecting* path p'' from s to v_m.

Now, consider this edge from v_k to v_m. Since this edge results in EAT at v_m, it implies that this edge results in earlier arrival at v_m than from v_{m-1}. In such a case, the thread for v_m would have processed this edge before the edge from v_{m-1} to v_m in Line 6. Consequently, it would have set the color of this edge to *green*. Now, when the thread for v processed the in-neighbors of v_m (Line 8), it process the edge from v_k to v_m first and since its color has already been set to green, the edge from v_{m-1} to v_m would never have been processed and its color set to green.

Thus, the thread for v would not have been able to identify the EAT path, p through v_{m-1}, which contradicts our assumption. □

Theorem 1: The algorithm correctly identifies the Earliest Arrival Time from the source vertex, s to all vertices.

Proof: Since the algorithm processes each vertex in parallel, establishing the proof for any arbitrary vertex suffices. Moreover, all the threads execute independently without any conflicts. The only dependencies that arise are when a thread (for a vertex, v) waits for other threads (Line 11) to set the color of edges to *red*/*green*.

Let us consider an arbitrary vertex, v in the input graph. We prove that the algorithm is able to correctly identify the EAT corresponding to the EAT path $p = (s, v_1, v_2, \ldots, v_m, v)$ from s to v. The proof is by induction on the length of the path, q between s and v on the path p.

Base Step: $q = 0$;

This represents the path of length 0 from s to itself and is straight forward.

Induction Step: Assuming that the theorem holds for $q = m$, we prove that the theorem holds for $q = m + 1$.

Let $p = (s, v_1, v_2, \ldots, v_{m+1}, v)$ be the EAT path from s to v having $m + 1$ intermediate vertices. We need to prove that the algorithm also identifies it as the EAT path from s to v.

For $q = m$, the proposed algorithm is able to identify an EAT path (by induction hypothesis) and therefore, by Lemma 1, $p' = (s, v_1, v_2, \ldots, v_{m+1})$ has to be the EAT path from s to v_{m+1}. We now show that the algorithm is able to identify EAT through p.

The thread of v scans its in-neighbors (Line 6) and picks the in-neighbor and the corresponding in-edge that returns the least arrival time at v. Since the in-edges of v is stored in a sorted order, they are scanned in ascending order of their arrival times at v.

Since p is the EAT path from s to v, it implies that either the edge from v_{m+1} to v returns the least arrival time at v or there exists another in-neighbor(s) of v, v_k such that the edge from v_k to v returns the least arrival time at v but the edge is not time-respecting. We discuss both the cases below.

Case 1: *The edge v_{m+1} to v results in the earliest arrival path at v.*

First, v_{m+1} and its corresponding in-edge to v is scanned (Lines 6-7). Next, the in-edges of v_{m+1} are scanned (Line 8). Since the algorithm has already found an EAT path from s to v_{m+1}, identified as p', the respective in-edge of v_{m+1} which is part of p' will be *green*. The if condition in Line 12 will be identified as true as the edge $v_{m+1} - v$ is *time-respecting* and the color of the in-edge of v_{m+1} corresponding to p' is also *green*. Consequently, the algorithm sets the color of $v_{m+1} - v$ edge as *green* (Line 15) denoting that it has found the EAT path to v.

Case 2: *There exists another in-neighbor(s) of v, v_k such that the edge from v_k to v returns the least arrival time at v but the edge is not time-respecting.*

In this case, the vertex, v_k will be processed earlier than v_{m+1}. However, the if condition in Line 10 will not be falsified and consequently, the thread will wait for all the in-edges of v_k to be colored red by other threads and finally will itself color the edge from v_k to v to red (Line 20). This will be repeated for every such vertex, v_k, and finally when v_{m+1} is processed, the edge from v_{m+1} to v will be colored green with the same argument as in Case 1, signifying the identification of EAT at v.

Thus, the theorem is proved. □

TABLE 1. The dataset.

Dataset	Vertices	Edges	levels(G')	Parallel Factor	Density
London	10.2K	5.26M	1994	2637.91	0.0505
Madrid	4.7K	1.99M	1574	1264.29	0.0901
Paris	411	18.2K	79	230.37	0.1077
Washington	5.9K	19.6K	68	288.23	0.0005

3) DETERMINING EARLIEST ARRIVAL PATHS

In addition to the earliest arrival times, the earliest arrival paths can also be identified with minor changes to the Algorithm 5. Note that there may exist multiple paths with the same EAT from source, s to a vertex, v, but due to the ordering imposed by our algorithm, it identifies a unique earliest arrival path from s to v for every vertex, v.

Let eAPath[i] represents a set that stores the earliest arrival path from source s to vertex i. For every vertex v, the earliest arrival paths from source s to v can be determined by inserting the following code snippet in Algorithm 5 after Line 15.

```
eAPath[v] = eAPath[w];
eAPath[v].add(u);
eAPath[v].add(v);
```

IV. EXPERIMENTS

We implemented all the parallel algorithms in CUDA and quantitatively compared those against the existing works. As per our best knowledge, the implementation of these works isn't available, thus, we implemented the existing algorithms by thoroughly following their ideas and applied them on experimental dataset to calculate execution times. Further, these executions times are the base of calculated speedup in our work.

A. EXPERIMENTAL SETUP

The machine used in our experiment comprises of 1.62 GHz NVIDIA Tesla K80 GPU with 12GB RAM and 2496 cores for the implementation of parallel algorithms. Besides, an INTEL core I3 CPU running at 2.00GHz with 8GB RAM and *gcc* version 5.4.0 is used to implement serial algorithms.

B. PARALLEL FACTOR

The parallel factor for a dataset indirectly indicates that how much it can be potentially parallelized. This statistical measure is accountable for the varying performance of algorithm(s) on different datasets.

TABLE 2. Execution time in milliseconds.

Dataset	Serial CSA	Parallel SPCS	RAPTOR	ESDG	Parallel CSA	Edge-version	Vertex-version	MTS-version	PBC-version	Selective-check
London	1304.68	901.22	835.71	791.97	623.17	510.55	689.94	549.36	252.22	198.73
Madrid	1109.36	823.91	785.23	690.82	551.59	415.21	525.59	429.84	184.22	173.66
Paris	576.87	556.58	565.46	372.09	295.68	281.52	323.59	237.03	105.11	84.22
Washington	527.37	517.23	511.07	340.35	221.15	213.54	227.15	163.66	99.81	87.46

For a temporal graph G, its parallel factor [6] is calculated by:-

$$\text{Parallel factor}, p(G) = |E|/l(G')$$

where, G' denotes the edge-scan-dependency graph of G, $|E|$ denote the no. of temporal edges in G, and $l(G')$ stands for no. of levels in G'.

C. DENSITY

The density of a graph indicates its sparseness/denseness. This measure is beneficial in gauging how many edges each thread may have to process.

The density of a graph G is defined by:-

$$\text{Density}, D(G) = \frac{|E|}{|V| * (|V| - 1)}$$

where, $|E|$ = no. of edges and $|V|$ = no. of vertices.

D. DATASET

The dataset used in the experiment consists of timetable information for 4 cities. The description of each dataset is as follows:-

E. RESULTS

We applied our algorithms to the dataset shown in Table 1. We ran 100 queries for each dataset on the proposed algorithms as well as on the existing algorithms and computed the average execution times, average speed-ups, and the amount of shared memory they occupy. The speedup of an algorithm x w.r.t. an algorithm y is calculated by the formula:-

$$\text{Speedup} = \frac{y_{ex}}{x_{ex}}$$

where,

x_{ex} = Execution time of algorithm x

y_{ex} = Execution time of algorithm y

Each query consists of a different source vertex. The average execution times of all the algorithms are presented in Table 2 and the amount of shared memory consumed by different algorithms are presented in Table 3. Similarly, the speedups of every proposed algorithm with existing algorithms are shown in Fig. 4. The Fig. 5 represents the performance of all the algorithms compared to serial CSA.

Our vertex-version algorithm proved faster than the existing serial algorithms but not on existing parallel algorithms. It is due to the multiple sequential iterations required to ensure accuracy. In the worst case, it goes to $O(|V|)$, where $|V|$ = no. of vertices. Its speedup is shown in Fig. 4a.

The next incremental algorithm, namely Multi-Time-scale version, utilizes shared memory for faster access on the in-edges and in-neighbors of a vertex. A Time scale is dedicated for each vertex whose function is very similar to the universal time scale used in the Time-scale version. It achieves speedup ranging from 1.52 to 3.22 w.r.t various algorithm on different datasets, as shown in Fig. 4b, thus, it shows a better performance than previous versions. The usage of the shared memory for faster access to the arrival times and the algorithm's capability to directly pick the least arrival time on a vertex to ensure its validity are the primary reasons for its improved performance.

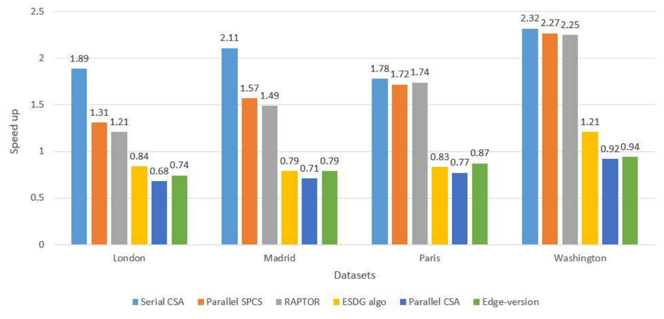
The Parallel-backward-check-version is an improved version of the Multi-Time-scale version that uses shared memory arrays to store the in-neighbors and the arrival times as key-value pairs instead of a time scale. It trimmed the need of scanning all the cells in the time scale and hence shows the average speedups ranging from 1.88 to 6.02 as shown in Fig. 4c. Another reason for its fast computation is the parallel scanning of shared memory instead of sequential scanning as in the previous version.

Our last algorithm, the Selective-check version, uses the sorted arrival times for each vertex which facilitates a faster analysis of the arrival times. As soon as any incoming edge is validated as the earliest arrival edge for a vertex v, it's been set for v and further scanning of the in-edges is stopped immediately. Consequently, the earliest arrival dependency of a vertex v on its in-neighbor u is highly reduced, hence the improved speedup. It shows the speedups ranging from 2.39 to 6.85 as shown in Fig. 4d which is an indicator of its decent performance.

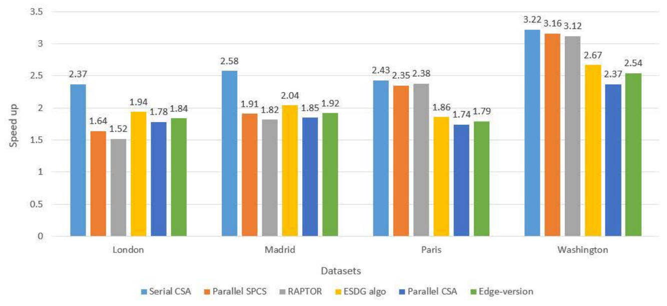
We didn't include the experimentation of the Time-scale version because of the absence of logic in it to address the *time-respectingness*. It is designed under the assumption that all the paths in the given network are *time-respecting*. Therefore, it is not useful particularly in real-life transport networks.

F. COMPARISON WITH SERIAL CSA

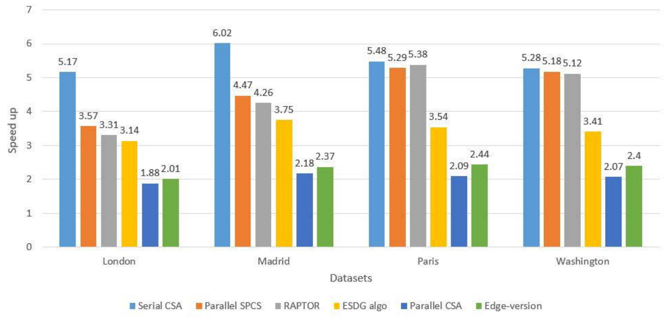
The performance of the proposed algorithms and the existing algorithms w.r.t. Serial CSA is shown in Fig. 5. The Selective-check version exhibits faster computation against all the mentioned algorithms. This happened due to its better ability to locate the least arrival times with minimum kernel overheads. The Parallel-backward-check algorithm also shows decent performance but not as fast as Selective-check. The reason is its tendency of being dependent



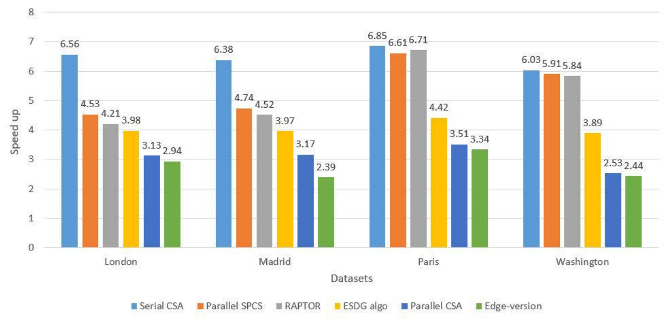
(a) Vertex-version



(b) Multi-Time-scale version



(c) Parallel backward-check-version



(d) Selective-check-version

FIGURE 4. Speedups of all approaches on various algorithm on all datasets.

TABLE 3. Bytes of shared memory occupied.

Dataset	Vertex-version	MTS-version	PBC-version	Selective-check
London	-	60.59MB	861.73KB	861.73KB
Madrid	-	22.92MB	326.04KB	326.04KB
Paris	-	209.66KB	29.12KB	29.12KB
Washington	-	225.79KB	31.36KB	31.36KB

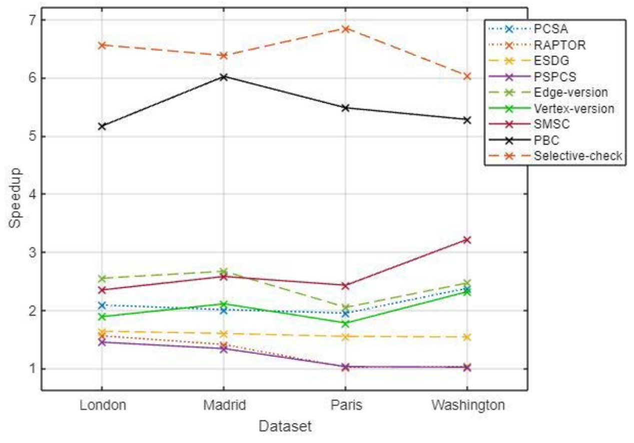


FIGURE 5. Speedups of various algorithms w.r.t. Serial CSA.

on the earliest arrival time of u for the calculation of the earliest arrival time of v, where u is the in-neighbor of v.

Among the existing algorithms, the Edge-version [5] algorithm shows the best speedup over others as shown in Fig. 5. The primary reason is the pruning of redundant connections, along with the techniques of creating connection-types, arithmetic progressions and clustering.

V. CONCLUSION

In this work, we have proposed a novel approach to solve the earliest arrival time problem. For proper route planning in a transport network, the arrival and departure times at every vertex should be carefully dealt with. Therefore, the set of arrival times corresponding to every vertex should be visualized as independent units. Our Selective-check algorithm makes the least number of comparisons on the best-case networks. A network is called best-case if $\forall v \in V$, if all arrival times at vertex v are less than the departure times from v. A network where no vehicles can be missed at any station is an example of the best-case network and hence, the earliest arrival time there can be quickly calculated. For average cases also, our Selective-check-version algorithm is faster than the existing works. The reason behind this achievement is the efficient access to the stored edges and the key idea used in the algorithm that the computation of EAT at a vertex, v need not be dependent on the EAT at the in-neighbors of v.

We hope that our analysis would prove useful in solving EAT problem for dynamic transport networks. Although the algorithms provided here have been designed for GPUs to exploit data parallelism, the ideas presented here are as well applicable to CPUs (with a few changes). We plan to implement an OpenACC version of the proposed parallel algorithms for performance portability across CPUs and GPUs. Additionally, it can be extended to multi-criteria problem solving with slight variations, for example, finding the earliest arrival route with the minimum cost of travelling.

ACKNOWLEDGMENT
The authors would like to thank the anonymous reviewers for their useful comments and editorial suggestions, which helped them to further improve the article.

REFERENCES

[1] J. Dibbelt, T. Pajor, B. Strasser, and D. Wagner, "Intriguingly simple and fast transit routing," in *Proc. Int. Symp. Exp. Algorithms*, 2013, pp. 43–54.
[2] B. B. Xuan, A. Ferreira, and A. Jarry, "Computing shortest, fastest, and foremost journeys in dynamic networks," *Int. J. Found. Comput. Sci.*, vol. 14, no. 2, pp. 267–285, Apr. 2003.
[3] D. Delling, T. Pajor, and R. F. Werneck, "Round-based public transit routing," *Transp. Sci.*, vol. 49, no. 3, pp. 591–604, Aug. 2015.
[4] D. Delling, B. Katz, and T. Pajor, "Parallel computation of best connections in public transportation networks," *ACM J. Experim. Algorithmics*, vol. 17, pp. 1–12, Jul. 2012.
[5] C. A. Haryan, G. Ramakrishna, R. Nasre, and A. D. Reddy, "A GPU algorithm for earliest arrival time problem in public transport networks," in *Proc. IEEE 27th Int. Conf. High Perform. Comput., Data, Anal. (HiPC)*, Dec. 2020, pp. 171–180.
[6] P. Ni, M. Hanai, W. J. Tan, C. Wang, and W. Cai, "Parallel algorithm for single-source earliest-arrival problem in temporal graphs," in *Proc. 46th Int. Conf. Parallel Process. (ICPP)*, Aug. 2017, pp. 493–502.
[7] H. Wu, J. Cheng, Y. Ke, S. Huang, Y. Huang, and H. Wu, "Efficient algorithms for temporal path computation," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 11, pp. 2927–2942, Nov. 2016.
[8] Y. Du, Q. Wu, Y. Zhao, X. Zhang, Y. Yao, and H. Xu, "A parallel time-varying earliest arrival path algorithm for evacuation planning of underground mine water inrush accidents," *Concurrency Comput., Pract. Exper.*, vol. 32, no. 11, p. e5644, Jun. 2020.
[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.
[10] *OpenMobilityData*. [Online]. Available: https://transitfeeds.com
[11] S. Maleki, D. Nguyen, A. Lenharth, M. Garzarán, D. Padua, and K. Pingali, "DSMR: A parallel algorithm for single-source shortest path problem," in *Proc. Int. Conf. Supercomput.*, Jun. 2016, pp. 1–14.
[12] P. J. Martín, R. Torres, and A. Gavilanes, "CUDA solutions for the SSSP problem," in *Proc. Int. Conf. Comput. Sci.* Berlin, Germany: Springer, 2009, pp. 904–913.
[13] S. P. Borgatti, "Centrality and network flow," *Social Netw.*, vol. 27, no. 1, pp. 55–71, Jan. 2005.
[14] G. J. Katz and J. T. Kider, "All-pairs shortest-paths for large graphs on the GPU," in *Proc. 23rd ACM SIGGRAPH Eurograph. Symp. Graph. Hardw.*, Jun. 2008, pp. 47–55.
[15] T. Shanmukhappa, I. W. Ho, C. K. Tse, X. Wu, and H. Dong, "Multi-layer public transport network analysis," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2018, pp. 1–5.
[16] K. Madduri, A. D. Bader, J. W. Berry, and J. R. Crobak, "An experimental study of a parallel shortest path algorithm for solving large-scale graph instances," in *Proc. 9th Workshop Algorithm Eng. Exp. (ALENEX)*, 2006, pp. 23–35.
[17] U. Meyer and P. Sanders, "Δ-stepping: A parallel single source shortest path algorithm," in *Proc. Eur. Symp. Algorithms*, vol. 1461. Heidelberg, Germany: Springer, 1998, pp. 393–404.
[18] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *J. Mach. Learn. Res.*, vol. 11, no. 3, pp. 985–1042, 2010.
[19] P. Yuan, C. Xie, L. Liu, and H. Jin, "PathGraph: A path centric graph processing system," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 3, pp. 2998–3012, Oct. 2016.
[20] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proc. 18th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, Feb. 2013, pp. 135–146.

SUNIL KUMAR MAURYA received the B.Tech. degree in computer science engineering from the Institute of Engineering and Rural Technology, Prayagraj, India, in 2020. He is currently pursuing the M.Tech. degree in information technology with IIIT Allahabad, India. His current research interests include parallel and distributed computing and high performance computing.

ANSHU S. ANAND (Senior Member, IEEE) received the B.Tech. degree in computer science and engineering from the Cochin University of Science and Technology, Kochi, in 2008, the M.Tech. degree in computer science engineering from the National Institute of Durgapur, West Bengal, India, in 2011, and the Ph.D. degree in computer science engineering from the Bhabha Atomic Research Centre, Mumbai, in 2019. He is currently an Assistant Professor with the Department of Information Technology, IIIT Allahabad, India. His research interests include parallel and distributed computing, high performance computing, parallel programming model design, programming languages, blockchain, and convergence of HPC and AI. He is a reviewer for many reputed peer-reviewed international journals and conferences.