GPU Accelerated Construction of Time Respecting Data Structure for Temporal Graphs

Animan Naskar*, Venkata Kalyan Tavva† and Subhasis Banerjee‡
* † Department of Computer Science and Engineering, Indian Institute of Technology Ropar, Punjab, India
Email: *2022csb1297@iitrpr.ac.in, †kalyantv@iitrpr.ac.in
‡ Shell India Markets Pvt. Ltd., Karnataka, India.
Email: subhasis.banerjee@shell.com

***Abstract*—The transformation of a temporal graph into its corresponding time-respecting graph (TRG) offers significant advantages for neighborhood search problems when compared to the traditional edge stream representations. However, constructing TRGs is a time-consuming process that can be substantially accelerated using GPU based parallelism. In this paper, we propose a novel highly parallel method to construct the time-respecting graph (TRG) from the edge list by leveraging GPU parallelism. For graphs of various sizes, ranging from 0.05 to 33 million edges, we achieve a speedup of up to 387x over the sequential variant on comparable hardware. We demonstrate that our method produces TRG in the standard CSR form that can be input to high-performance graph analytics libraries such as nvGRAPH. The TRG produced is also optimized for a minimal number of vertices and edges. Our method also provides support for efficient real-time dynamic updates to the temporal graph.**

***Keywords*—** Contact sequence temporal graphs, time-respecting graph, data-structure, parallel, GPU, CUDA

I. Introduction

Temporal graphs represent networks where edges can be taken only at specific timestamps. Unlike static graphs, temporal graphs capture the temporal nature of edges, thereby offering a more realistic representation of real-world systems [1]. For instance, consider a flight network where vertices represent airports and edges denote flights scheduled at specific times. In such a temporal graph, the existence of an edge between two airports depends not only on their connectivity but also on the specific time when a flight operates between them. A survey of temporal networks [2] shows their diverse real-world applications, spanning from disease spread study, communication networks, road networks, etc.

In this paper we look at Contact Sequence Temporal Graphs (CSG). Edges can be taken at discrete time instances. A CSG has edges of the form $< u, v, t, w >$ where u is the from-vertex and v is the to-vertex. t is its timestamp at which the edge can be taken and w $(w \geq 0)$ is the time it takes to get from u to v along this edge. If t is the departure time at u, $t + w$ is the arrival time at v. The CSG can be represented as an edge stream or transformed into an equivalent Time-Respecting Graph (TRG) [1]. The edge stream is a list of all the edges in increasing order of their timestamps. The TRG is a graph in which all paths are time-respecting (also called "journeys") [3]. A time-respecting path from u to v requires that the departure time from each vertex along the path is greater than or equal to the arrival time at that vertex.

Temporal graph algorithms follow different approaches depending on the underlying graph representation. Prior work [1] has shown that algorithms optimized for TRG perform better than those optimized for edge stream representation for solving neighborhood search problems. In the edge stream representation, edges are scattered throughout the graph and their information is stored without regard to the spatial locality of vertices. Consequently, algorithms applied to edge stream representations must perform a full pass over all the edges [4] for every query, even for neighborhood search problems. Conversely, in TRGs, vertices are organized based on their spatial and temporal configurations that can enhance algorithmic efficiency for such types of problems.

To the best of our knowledge, thus far, only sequential implementations for creating TRG exist in the literature [1]. In this paper we propose a novel highly parallel GPU based method to create the TRG. On various real-world graphs ranging from 0.05 to 33 million edges, we achieve a speedup of up to 387× over the sequential variant on comparable hardware. Reduction in the time to create the TRG further reduces the time required to begin analysis and apply graph algorithms. In addition, we incorporate certain reduction techniques while building the TRG that further benefit a set of graph applications. Prior work [5] has demonstrated that algorithms applied to the reduced TRG perform comparably if not better than those applied to the edge stream for global search problems, such as source-to-all-vertices problems.

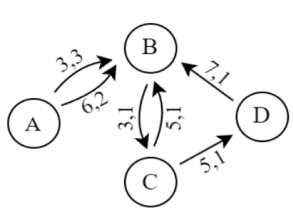
II. Existing TRG Variants

An example of a contact sequence temporal graph $G = (V, E)$ is shown in Figure 1a with four vertices. Each edge in this graph is marked with two values, namely, $timestamp$ and $weight$. There is a possibility of multiple edges existing between two vertices, with different timestamps. For example, in the graph G, we can observe two edges originating from A to B, with timestamps '3' and '6'. Figure 1b shows the edge stream representation of the example graph under consideration. Each entry in this representation provides information about a single edge. An edge is of the form $< u, v, t, w >$ where u is from-vertex, v is to-vertex, t is timestamp at which the edge can be taken and w is the weight or simply time taken to traverse the edge. For example, the first entry in the edge stream shown in Figure 1b conveys that there is an edge from vertex B to vertex C at time '3' and with a weight of '1'. In case the weight is representing the traversal time, then the time by which one can reach C is '4'. Next we look at the various TRG representations proposed over the years.

A. TRG-Wu

Figure 1c shows the TRG-Wu [1] representation of the temporal graph G. It can be observed that all the paths are time respecting. Consider vertex A of G. Let,

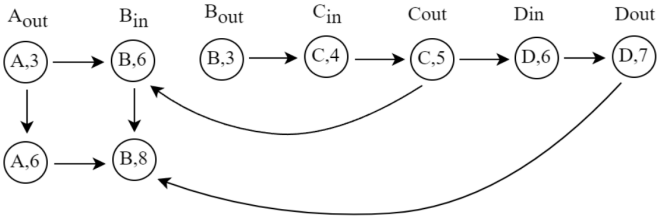
$$T_{in}(A) = \{t \mid (*, A, *, t) \in G\}$$
$$T_{out}(A) = \{t \mid (A, *, *, t) \in G\}$$
$$V_{in}(A) = \{(A, t) \mid t \in T_{in}(A)\}$$
$$V_{out}(A) = \{(A, t) \mid t \in T_{out}(A)\}$$



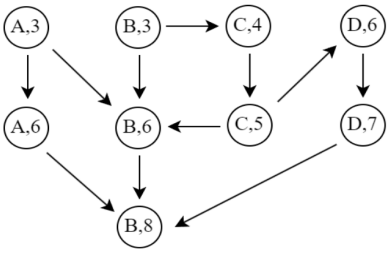
(a) Example Temporal Graph G

B, C, 3, 1
A, B, 3, 3
C, B, 5, 1
C, D, 5, 1
A, B, 6, 2
D, B, 7, 1

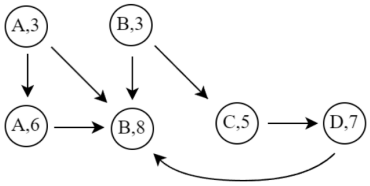
(b) Edge Stream(G)



(c) TRG-Wu(G)



(d) TRG-GPU(G) un-reduced



(e) TRG-GPU(G) reduced

Fig. 1: Temporal Graph and its representation example.

$T_{in}(A)$ is a set to all distinct times of arrival to the vertex A and $T_{out}(A)$ is a set of all distinct times of departure from the vertex A. Hence, vertex A corresponds to two set of vertices $V_{in}(A)$ and $V_{out}(A)$ in the TRG-Wu, wherein $V_{in}(A)$ and $V_{out}(A)$ are sets of vertices corresponding to distinct arrival and departure times, respectively. $< u, v, t, w >$ corresponds to an edge from $(u, t) \in V_{out}(u)$ to $(v, t+w) \in V_{in}(v)$ in the TRG-Wu. The TRG-Wu also has a few additional edges as explained below:

1) Let $V_{in}(A) = \{(A, t_1), (A, t_2), \ldots, (A, t_k)\}$ where vertices are ordered in decreasing timestamp. Edges exist from $(A, t_{i+1}) \in V_{in}(A)$ to $(A, t_i) \in V_{in}(A)$ for all $1 \leq i \leq k-1$. Similarly, edges exist between $V_{out}(A)$ vertices.
2) For every vertex $(A, t_{in}) \in V_{in}(A)$, an edge to $(A, t_{out}) \in V_{out}(A)$, where $t_{out} = \min\{t \mid t \in T_{out}(A), t \geq t_{in}\}$.

The above representation is generalized for all the vertices and edges. The additional edges are called *wait-edges* and traversing these implies waiting at a vertex, possibly to take an edge at a higher departure time. For example, in Figure 1c, edges between $(A, 3)$ and $(A, 6)$, $(B, 6)$ and $(B, 8)$, etc., are *wait-edges*.

An improvement on TRG-Wu was proposed by Gheibi et al. [5] wherein the authors reduce the TRG by eliminating the redundant vertices and *wait-edges*. The reduced TRG is shown to perform better than TRG-Wu for global search problems.

B. TRG-GPU

This TRG representation is created by our GPU parallel method, and is henceforth referred to as TRG-GPU. It is in reduced form unless real-time updates are allowed where we leave it un-reduced. Due to our optimized creation method, our un-reduced form is a slight variation of the TRG-Wu as explained next.

1) Un-reduced TRG

Referring to Figure 1d, let

$$V(A) = V_{in}(A) \cup V_{out}(A)$$
$$T(A) = T_{in}(A) \cup T_{out}(A)$$

where, $V_{in}(A)$, $V_{out}(A)$, $T_{in}(A)$, $T_{out}(A)$ are defined as before in the case of TRG-Wu and $V(A)$ is a set of all vertices in un-reduced TRG corresponding to vertex A of temporal graph G. Note that, unlike in TRG-Wu, where $(A, t) \in V_{in}(A)$ and $(A, t) \in V_{out}(A)$ are differentiated, in this TRG representation there is only one corresponding vertex $(A, t) \in V(A)$. We generalize the same to all other vertices.

The edge set is constructed similarly to TRG-Wu:

1) Corresponding to every edge of the form $< u, v, t, w >$ of the temporal graph G, an edge exists from $(u, t) \in V(u)$ to $(v, t + w) \in V(v)$ in the TRG.
2) Let $V(A) = (A, t_1), (A, t_2), \ldots, (A, t_k)$ in which vertices are in increasing order of t. Edges exist from (A, t_i) to (A, t_{i+1}) $\forall\ 1 \leq i \leq k-1$. These are wait-edges of the TRG-GPU.

2) Reduced TRG

Considering Figure 1e, our TRG-GPU construction method creates the reduced TRG variant proposed by Gheibi et al. [5]. Consider reduction from Figures 1c, 1d and 1e. To determine if a vertex (A, t) should be reduced, we need to check for two conditions:

1) **Outgoing Edge Condition**: The vertex (A, t) must have exactly one outgoing edge.
2) **Successor Condition**: It must be followed by another vertex in the list $V(A) = \{(A, t_1), (A, t_2), \ldots, (A, t_k)\}$.

If a vertex (A, t) satisfies these conditions it is eligible for reduction. The reduction process involves merging (A, t) with the vertex immediately following it in the $V(A)$. Assume (A, t_i) gets reduced, then it is merged with (A, t_{i+1}). Such merging involves,

1) All incoming edges to (A, t_i) are redirected to (A, t_{i+1}).
2) The outgoing edge from (A, t_i), that is also a wait edge to (A, t_{i+1}), is removed.

This check is performed for all vertices, and reducible vertices are removed to create the reduced form. Our method directly constructs the reduced form, bypassing the creation of the un-reduced form.

TABLE I: Vertices and their mapping for the example temporal graph G.

	0	1	2	3	4	5	6	7	8	
	A, 3	A, 6	B, 3	B, 6	B, 8	C, 4	C, 5	D, 6	D, 7	0
	0, 3	0, 6	1, 3	1, 6	1,8	2, 4	2, 5	3, 6	3, 7	1
map[] →	0	3	6	9	11	13	14	21	22	2

III. Construction of TRG-GPU

This section explains our novel approach to construct the reduced TRG-GPU data structure from the edge list of a given temporal graph, using GPU to exploit maximum parallelism. We first explain some critical aspects of the method and then dive into the parallel implementation.

CSR Format for TRG: Consider a vertex A in a temporal graph G, it corresponds to the set of vertices $V(A)$ in the TRG. The outgoing edges from vertex A get distributed over multiple vertices $V(A)$ in the TRG. As a result, the corresponding TRG is generally sparse. Hence, we make use of the Compressed Sparse Row (CSR) [10] format instead of the adjacency matrix.

Another memory efficient approach is to allocate space dynamically for the neighbors of each vertex separately. However dynamic memory allocation calls are expensive on a GPU and a single call for each vertex would consume a lot of time in case of a large graph. In contrast, the CSR format allocates space for the neighbors of all vertices with a single `malloc` call, making it an ideal choice for large/sparse graphs. However, handling graph updates with an underlying CSR data structure involves expensive memory operations. Fortunately, when using GPU acceleration, we typically deal with large temporal graphs where updates are small and infrequent.

Vertex Representation: Consider vertex (A, t). Treating it as a pair of integers is expensive. Hence, we assign it a unique value, A_t, defined as,

$$A_t = A \times \Delta T + (t - t_{min}) \tag{1}$$

In the prior figures, as a generic example we use characters (ex. A, B, C, D) for vertices. In practice, the vertices are represented as values (ex. 0, 1, 2, 3). Hence, in Equation 1, the corresponding value of the vertex is used for calculating A_t.

Let us now define two parameters as below:
t_{min} = minimum timestamp of a vertex
t_{max} = maximum timestamp of a vertex
We can now define,

$$\Delta T = t_{max} - t_{min} + 1 \tag{2}$$

CSR Arrays: As mentioned earlier CSR involves the $cols$[] & $rows$[] arrays. To construct $rows$[] we first calculate degrees of all the vertices and store them in another array, say, *deg*[].

Consider vertex (A, t), its degree is stored as $deg[A_t]$ because A_t is unique to the vertex. Then $rows[A_t]$ is offset of the first neighbor of vertex (A, t) in $cols$[]. Similarly, if (B, t+w) is neighbor and it corresponds to B_t+w, we store value B_t+w in $cols$[].

Memory Inefficiencies: Table 1 shows the vertices of the temporal graph, Figure 1(a). Row 2 shows their corresponding values. Columns are sorted in ascending order of the vertex values. Using these values directly for indexing arrays deg[] and $rows$[] will result in many empty indices because the values are not necessarily contiguous. Instead, we use the column number of each vertex in Table I for indexing purposes as these are contiguous and thus eliminate gaps.

In temporal graph $G = (V, E)$, earlier we had to index arrays deg[] and $rows$[] for all $|V| \times \Delta T$ possible vertices. Now we index only the vertices that are actually part of the temporal graph. Often temporal graphs with a few vertices can span over a large ΔT time-range. The unoptimized approach would create arbitrarily high memory demand due to in-efficient space usage.

Our method involves sorting vertex values. We call the sorted array map[]. Consider storing the degree of vertex (A, t) in array deg[].

1) We can look-up map[] for A_t by means of $O(logn)$ lower bound binary search in map[].
2) We denote look-up return value as A_t. Then, $deg[A_t]$ is the degree of (A, t).

Parallel Implementation: Our implementation uses the CUDA parallel computing platform [6] for NVIDIA GPUs. We can divide our implementation methodology into five major parts, and explain each of them next as Algorithms 1, 3, 4, 5 and 6.

Algorithm 1: Construction of map[] array from input edge list.

Input: Text file containing temporal graph edge list.
Output: Array that stores edges $edge\_list$[]; map[] array; number of vertices of TRG, $num\_vertices$.

1 Input text file is parsed line by line and edges of the form $< u, v, t, w >$ are appended at the end of $edge\_list$ vector. The vector is then copied to device memory.
2 **parallel for** *each edge* $< u, v, t, w > \in edge\_list$ **do**
3 $\quad threadId = blockIdx.x \times blockDim.x + threadIdx.x$
4 $\quad map[threadId \times 2] = u_t$
5 $\quad map[threadId \times 2 + 1] = v_{t+w}$
6 map.**sort_unique()**
7 $num\_vertices = map.len()$

Algorithm 2: $sort\_unique()$

Input: Array with possible duplicates.
Output: Sorted array without duplicates.

1 Call thrust::sort() [7] `/* Array is sorted in parallel */`
2 Call thrust::unique() [7] `/* All duplicates are removed as array is sorted making duplicates consecutive. */`

Algorithm 1 constructs an array of all TRG vertices from the input edge list by reading it line by line and calculating vertex values

as per Equation 1. In order to remove the duplicates and sort the resultant $map[]$, as shown in Algorithm 2 we use two `thrust` library functions, namely, *sort()* and *unique()* for their efficient parallel implementation.

Algorithm 3: Identification and removal of reducible vertices.

Input: $map[]$ array and $edge\_list[]$ array.
Output: $map[]$ array after removal of reducible vertices.

1 Initialize $out\_edge[num\_vertices] = \{0\}$
2 **parallel for** *each edge* $< u, v, t, w > \in edge\_list$ **do**
3 $\quad u^t = map.lower\_bound(u_t)$
4 $\quad$ **Atomic-modify** $out\_edge[u^t] = 1$
5 **parallel for** *each entry* $A_t \in map[]$ **do**
6 $\quad A^t = map.lower\_bound(A_t)$
7 $\quad$ **if** $out\_edge[A^t] == 0$ **then**
8 $\quad\quad$ **if** $A^t + 1 < num\_vertices$ **then**
9 $\quad\quad\quad map[A^t] = -1$
10 All -1s are parallelly removed from the $map[]$ array using $thrust$::$copy\_if$ [7].
11 **free**(out_edge)

Algorithm 3 removes reducible vertices from $map[]$. The reducible vertices are removed even before the $cols[]$ and $rows[]$ arrays are even initialized. When an incoming edge to a reduced vertex has to be placed, the reduced-vertex (i.e. to-vertex) gets looked up in the $map[]$. The function $lower\_bound()$ (in Lines 3 and 6) performs look-up into sorted array in $\mathcal{O}(\log n)$ time, where n is length of the array; and returns the index. As we use $lower\_bound$ binary search for look-ups and the vertex is already removed, the successor vertex (next vertex) is returned. So, in effect all the incoming edges get directed to the successor vertex. Also note that removing the reducible vertices beforehand prevents inclusion of their *wait edges* in the reduced TRG. Our approach is efficient because we can directly create the reduced TRG.

Algorithm 4: Degree calculation for all the vertices.

Input: $map[]$ array and $edge\_list[]$ array.
Output: Array that stores degree of each vertex $deg[]$ and $edge\_pos[]$ array.

1 Initialize $deg[num\_vertices] = \{0\}$
2 **parallel for** *each entry* $A_t \in map[]$ **do**
3 $\quad A^t =$ map.**lower_bound**(A_t)
4 $\quad$ **if** $A^t + 1 < num\_vertices$ **then**
5 $\quad\quad$ **if** $map[A^t + 1]/\Delta T == A_t/\Delta T$ **then**
6 $\quad\quad\quad deg[A^t]++$
7 **parallel for** *each edge* $< u, v, t, w > \in edge\_list$ **do**
8 $\quad threadId =$
$\quad\quad blockIdx.x \times blockDim.x + threadIdx.x$
9 $\quad A^t =$ map.**lower_bound**(A_t)
10 $\quad edge\_pos[threadId] =$ **atomicAdd**$(deg[A^t], 1)$

Algorithm 4 is next used to determine the degrees of all the vertices in $map[]$, storing the resultant degrees in another, namely, $deg[]$. *atomicAdd (addr, val)* (Line 10) atomically adds the value of val to the double located at the given address $addr$. It returns the original value stored at the address before the addition.

Inserting neighbors of a vertex in $cols[]$ array and incrementing its degree are both sequential tasks. However, some parallelism can be introduced when neighbors of different vertices are involved in either case. Ideally we would like to insert neighbors of a vertex in the $cols[]$ array while incrementing its degree. However, $cols$ array is not initialized at this stage. So, the i^{th} thread corresponding to the i^{th} edge $< u, v, t, w >$ increments the degree of (u, t) and stores an offset in $edge\_pos[i]$. Later, when the $cols[]$ array is initialized, the i^{th} thread again examines the i^{th} edge and inserts the neighbor vertex $(v, t+w)$ at index $rows[u^t] + edge\_pos[i]$ where $edge\_pos[i]$ is the offset stored earlier.

Algorithm 5: Calculation of $rows[]$ array

Input: Array $deg[]$.
Output: Array $rows[]$, i.e., exclusive prefix sum array of $deg[]$.

1 Exclusive prefix sum array, $rows[]$ is parallelly calculated for $deg[]$ array using $thrust :: exclusive\_scan$ [7].

Once the degrees are determined, we next perform prefix sum of $degrees[]$ to obtain $rows[]$ array using Algorithm 5. We use another function from `thrust` library, *exclusive_scan* for this purpose. As explained earlier, this prefix sum calculation is very parallelized and efficient. Lastly, the $cols[]$ array is filled by examining the edge list again and offsets stored in $rows[]$ array using Algorithm 6.

Algorithm 6: Insertion of neighbor vertices in cols[] array.

Input: Arrays $edge\_list[]$, $rows[]$ and $edge\_pos[]$.
Output: Array $cols[]$ after neighbor vertices are inserted.

1 Initialize $num\_edges =$
$\quad rows[num\_vertices - 1] + deg[num\_vertices - 1]$
2 Initialize $cols[num\_edges] = \{-1\}$
3 **parallel for** *each edge* $< u, v, t, w > \in edge\_list$ **do**
4 $\quad threadId =$
$\quad\quad blockIdx.x \times blockDim.x + threadIdx.x$
5 $\quad u^t =$ map.**lower_bound**(u_t)
6 $\quad v^{t+w} =$ map.**lower_bound**(v_{t+w})
7 **free**($edge\_pos$)
8 **parallel for** *each entry* $A_t \in map[]$ **do**
9 $\quad A^t =$ map.**lower_bound**(A_t)
10 $\quad$ **if** $cols[rows[A^t]] == -1$ **then**
11 $\quad\quad cols[rows[A^t]] = A^t + 1$

IV. Dynamic TRG-GPU

In this section, we propose a parallel method for handling real-time queries to update the TRG-GPU on a GPU. Although the implementation of a fully-dynamic data structure for a dynamically varying TRG is complex, we present a theoretical outline of an efficient approach. Our goal is to showcase the modular design of the proposed TRG-GPU and provide a foundation for future research

and development. We next describe some critical aspects considered in the design of our approach.

Issues with updating reduced TRG: Let vertex (A,t) be reducible and $(A,t+w)$ be the successor vertex. Consider a case in which TRG has been constructed and in an update, an outgoing edge is added to vertex (A,t). This necessitates undoing the reduction of (A,t), as it no longer meets the reducibility criterion. However, our approach deletes the reducible vertex (A,t) from $map[]$ at the beginning of TRG creation, making it inefficient to undo this reduction. Hence, if a dynamic TRG is needed, it is not reduced while construction.

Updating $map[]$ and look-up: Our method parallely appends new vertices to the end of the $map[]$. This leads to a sorted segment (existing vertices) and an unsorted segment (new vertices). We store the length up to which $map[]$ is sorted. Look-up for a new vertex involves launching parallel threads for every entry in the unsorted segment. Each thread compares its entry to the desired vertex. If a match is found, it updates a passed parameter to the kernel with the position.

Note: Look-up in the old segment always returns a position even in the case it is absent as it is a lower-bound search. Hence, only if a vertex is not found in the new segment, a look-up in the old segment is required.

Inserting new vertices anywhere except the end will lead to shifting of other vertices in effect changing their look-up values. Look-up value must not change as it indexes the $rows[]$, $degrees[]$ arrays. Hence sorted order of $map[]$ cannot be preserved once TRG is updated and needs to be rebuilt after every new insert.

Adding Edges: The CSR data structure has no gaps. To add more neighbors for a vertex, it is deleted and reinserted with updated edges, relocating its neighbors to the end of the cols array. This process is done in parallel for each added edge.

Introducing new vertices may also cause implicit addition and redirection of wait-edges. Consider vertex (A,t) which does not have a wait-edge. However an update may introduce a vertex $(A,t+w)$ to the graph which implicitly creates a need for a wait-edge from (A,t) to $(A,t+w)$. This is the same as adding neighbors to the vertex and dealt with similarly.

Deleting Edges : The edge deletion process consists of two main steps: masking and removal. For each edge $< u,v,t,w >$ to be deleted, first, identify the entry corresponding to $(v,t+w)$ among the neighbors of (u,t) in $cols[]$ and set it to -1 in parallel. Then, decrement the entry corresponding to (u,t) in $degrees[]$ by 1. If the degree becomes 0, set entry corresponding to (u,t) as -1 in the map.

After all edges are processed, remove all -1 entries from $cols[]$, $map[]$, and 0 entries from $deg[]$ using $thrust :: copy\_if$. Finally, reconstruct $rows[]$ from $deg[]$ using a parallel exclusive scan. This method is particularly efficient for batched updates.

V. Complexity Analysis

Table II summarizes the work and time complexities of our methodology. Assume temporal graph $G=(V,E)$. Let $\overline{G}=(\overline{V},\overline{E})$ denote the corresponding reduced TRG. Then,

$$|\overline{V}| \leq 2|E| \tag{3}$$

$$\begin{aligned} |\overline{E}| &= |E| + \Sigma_{A\in V}(|V(A)|-1) \\ &= |E| + |\overline{V}| - |V| \leq 3|E| - |V| \end{aligned} \tag{4}$$

Assume there are N parallel cores available.

TABLE II: Time Complexity.

Algorithm	Line#	Sequential	Parallel
Algo. 1	2	$\Theta(\|E\|)$	$\Theta(\|E\|/N)$
Algo. 2	1	$\mathcal{O}(\|E\| \times \log\|E\|)$	$\Omega(\|E\| \times \log\|E\|/N)$
	2	$\Theta(\|E\|)$	$\Omega(\|E\|/N)$
Algo. 3	2	$\Theta(\|E\| \times \log\|\overline{V}\|)$	$\Omega(\|E\| \times \log\|\overline{V}\|/N)$
	5	$\Theta(\|\overline{V}\|)$	$\Theta(\|\overline{V}\|/N)$
	9	$\Theta(\|\overline{V}\|)$	$\Omega(\|\overline{V}\|/N)$
Algo. 4	2	$\Theta(\|\overline{V}\|)$	$\Theta(\|\overline{V}\|/N)$
	7	$\Theta(\|E\|)$	$\Omega(\|E\|/N)$
Algo. 5	1	$\Theta(\|\overline{V}\|)$	$\Omega(\|\overline{V}\| \times \log\|\overline{V}\|/N)$
Algo. 6	2	$\Theta(\|\overline{E}\|)$	$\Theta(\|\overline{E}\|/N)$
	3	$\Theta(\|\overline{E}\| \times \log\|\overline{V}\|)$	$\Theta(\|\overline{E}\| \times \log\|\overline{V}\|/N)$
	7	$\Theta(\|\overline{V}\|)$	$\Theta(\|\overline{V}\|/N)$

A. Overall time-complexity of sequential variant is given as:

$$= \Theta(|\overline{E}| \times \log|\overline{V}|) + O(|E| \times \log|\overline{E}|) \tag{5}$$

Substituting Equations 3 and 4 we get an upper bound:

$$= \Theta(|E| \times \log(|E|+|V|) + \mathcal{O}(|E| \times \log|E|) \tag{6}$$

B. Overall time-complexity of parallel variant is given as:

$$= \Theta(|E| \times \log|\overline{V}|/N) + \Omega(|E| \times \log|E|/N) \tag{7}$$

Substituting Equations 3 and 4 we get an upper bound:

$$= \Theta(|E| \times \log(|E|+|V|)/N) + \mathcal{O}(|E| \times \log|E|/N) \tag{8}$$

After substituting Equations 3 and 4, we get memory complexity of our method to be $\mathcal{O}(|V|)+\mathcal{O}(|E|)$. The memory complexity of each individual part of the proposed method is provided in Table III.

TABLE III: Memory Complexity.

Algorithm	Memory
Algo. 1 & 2	$\mathcal{O}(\|E\|)$
Algo. 3	$\mathcal{O}\|\overline{V}\|$
Algo. 4	$\mathcal{O}\|\overline{V}\|$ + $\mathcal{O}(\|E\|)$
Algo. 5	$\mathcal{O}\|\overline{V}\|$
Algo. 6	$\mathcal{O}(\|\overline{E}\|)$

VI. Experimental setup and results

In this section we evaluate the performance of our method and compare it against the sequential counterpart. The sequential implementation is run on a CPU with AMD EPYC 7V13 Processor with frequency $2.4GHz$, 64 Cores and 226 GB RAM. The sequential approach is single threaded and we use the *g++* compiler with the *O3* optimization. The GPU used is NVIDIA A100 with frequency 1.06 GHz, consisting of 6912 Cores and 80GB device memory. For the parallel approach, we use the *nvcc* compiler with the *O3* optimization.

We use six datasets from The KONECT Project [9], namely, *ia-reality-call* (D1), *munmun_digg_reply* (D2), *tech-as-topology* (D3), *sx-mathoverflow* (D4), *ca-cit-HepPh* (D5) and *out.flickr-growth* (D6). Table IV lists the graph datasets used for experimentation. These datasets were chosen because of their varied characteristics.

TABLE IV: Details of the datasets used in experimentation.

Dataset	$\|V\|$	$\|E\|$	$\|\overline{V}\|$	$\|\overline{E}\|$	ΔT
D1	6809	52050	52273	103854	9159564
D2	30398	87627	89016	173356	1306497
D3	34761	171403	60944	230216	2016006
D4	24818	506550	506511	1013040	203069369
D5	16959	2322259	176134	2498376	245980802
D6	2302925	33140017	5343174	38482938	17017202

$G = (V, E)$ denotes the temporal graph, $\overline{G} = (\overline{V}, \overline{E})$ denotes the reduced TRG, and ΔT to denote the timespan of the temporal graph.

A. Results

The following Table V and VI report time taken by the sequential method and our parallel GPU-based method for TRG construction, respectively. To gain additional insights we provide timings of each of the major functions discussed earlier for both the methods.

TABLE V: Average time (in ms) taken for creating reduced TRG on CPU.

Dataset	Algo. 1 & 2	Algo. 3	Algo. 4	Algo. 5	Total
D1	27.65	4.16	0.73	7.24	39.78
D2	49.37	8.40	0.96	13.30	72.03
D3	85.30	13.70	1.12	26.90	126.74
D4	391.04	44.03	2.86	79.65	517.57
D5	423.11	59.06	4.48	195.30	681.95
D6	25016.1	1437.75	74.39	5999.55	32527.79

TABLE VI: Average time (in ms) taken for creating reduced TRG on GPU.

Dataset	Algo. 1 & 2	Algo. 3	Algo. 4	Algo. 5	Total
D1	1.02	0.04	0.14	0.05	1.25
D2	1.19	0.10	0.19	0.05	1.54
D3	1.52	0.12	0.19	0.07	1.90
D4	2.61	0.22	0.40	0.16	3.39
D5	6.45	0.32	0.28	0.26	7.31
D6	72.11	3.79	0.49	7.58	83.96

The observed increase in experiment runtimes with the rising number of edges aligns well with the theoretical time complexity derived in earlier section. We also observe increasing performance gain with increasing input graph size in Table VII. The trend is due to the degrading CPU performance as it is significantly influenced by its cache hierarchy. When processing very large graphs, the CPU cache can be of limited use, leading to cache thrashing that negatively impacts the sequential performance. This is the reason we observe a drastic increase in the parallel performance gain over sequential from smaller graphs like *tech-as-topology* (D1) to larger graphs like *ca-cit-HepPh* (D6). Performance gain is defined as the ratio of fraction of time taken by the parallel method when compared to the sequential method.

Our experimental results show that the parallel approach can reduce execution time by up to $387\times$ compared to the sequential approach when constructing TRGs of graphs with up to 33 million edges (D6). Additionally, to evaluate the scalability of our method, we perform experiments on two large synthetic graphs containing 238 million and 585 million edges and observe remarkable performance gains of $1536\times$ and $1702\times$, respectively. These experiments show that the proposed method is efficiently utilizing the available parallel in GPUs.

TABLE VII: Performance gain of parallel implementation over sequential.

Dataset	Performance gain
D1	31.9
D2	46.7
D3	66.6
D4	152.9
D5	93.4
D6	387.4

It is worthy to be noted that the performance gains are highly dependent on the CPU and GPU configurations. When experiments were conducted on a CPU running at $2.2GHz$ and NVIDIA Tesla T4 GPU with frequency $1.59GHz$, 2560 Cores, we observed performance gains of 16.1, 25.2, 51.4, 85.4, 72.3 and 138.9 for datasets D1, D2, D3, D4, D5 and D6, respectively. But both the sets of experiments demonstrate the efficiency of the proposed parallel implementation of TRG creation.

Given that the output TRG of the proposed parallel method is in the standard CSR format, high-performance graph analytics libraries such as nvGRAPH [11] can use it directly without any additional preprocessing and work on the popular graph algorithms in parallel.

VII. Conclusion

In this paper, we presented an efficient GPU-based parallel method for constructing the time respecting TRG data structure from the edge list of a temporal graph. The resulting TRG is output in the standard Compressed Sparse Row (CSR) format. The TRG constructed is superior to the traditional edge stream representation on problems requiring only local information and highly competitive for single-source all-destinations problems [5]. When using a TRG representation, many temporal graph problems, such as the 'foremost path,' essentially become variations of the vertex reachability problem — determining whether one vertex can be reached from another. This can be easily parallelized in a manner similar to BFS on GPUs [1]. Our solution is well-suited for very large real-world temporal networks, like flight networks and telecommunication networks. Our experimental results demonstrate substantial time reductions, achieving improvements of up to two-three orders of magnitude compared to existing sequential methods.

Acknowledgments

The authors thank the anonymous reviewers for their constructive feedback that helped to improve the draft. This work is supported by Shell India Markets Pvt. Ltd. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Shell India.

References

[1] Huanhuan Wu, James Cheng, Yiping Ke, Silu Huang, Yuzhen Huang and Hejun Wu, "Efficient Algorithms for Temporal Path Computation", IEEE Transactions on Knowledge and Data Engineering,2016, Vol. 11, pp. 2927-2942, DOI: 10.1109/TKDE.2016.2594065.

[2] Petter Holme and Jari Saramäki, “Temporal Networks”, Physics Reports, Elsevier BV, 2012, Vol. 519, pp. 97–125, DOI: 10.1016/j.physrep.2012.03.001.

[3] David Kempe, Jon Kleinberg and Amit Kumar, “Connectivity and Inference Problems for Temporal Networks”, Journal of Computer and System Sciences, 2002, Vol. 64, pp. 820-842,DOI: https://doi.org/10.1006/jcss.2002.1829.

[4] B. Bui Xuan, A. Ferreira and A. Jarry, “Computing shortest, fastest, and foremost journeys in dynamic networks”, International Journal of Foundations of Computer Science, 2003, Vol. 14, pp. 267-285, DOI:10.1142/S0129054103001728.

[5] S. Gheibi, T. Banerjee, S. Ranka and S. Sahni, “An Effective Data Structure for Contact Sequence Temporal Graphs,” 2021 IEEE Symposium on Computers and Communications (ISCC), 2021, pp. 1-8, DOI: 10.1109/ISCC53001.2021.9631469.

[6] “NVIDIA CUDA Toolkit”, 2024. Available: https://developer.nvidia.com/cuda-toolkit

[7] “Thrust: The C++ Parallel Algorithms Library”, 2024. Available: https://nvidia.github.io/cccl/thrust/

[8] “CUDA C++ Programming Guide”, 2024. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[9] Jérôme Kunegis. “KONECT: the Koblenz network collection”, 2013, 22nd ACM International Conference on World Wide Web (WWW ’13 Companion), pp. 1343–1350. DOI: https://doi.org/10.1145/2487788.2488173, Available: http://konect.cc/

[10] Thorsten Blaß and Michael Philippsen, “Which Graph Representation to Select for Static Graph-Algorithms on a CUDA-capable GPU”, 2019, ACM Proceedings of the 12th Workshop on General Purpose Processing Using GPUs, pp.22-31, DOI: 10.1145/3300053.3319416.

[11] nvGraph API. 2024. Available: https://docs.nvidia.com/cuda/archive/10.1/nvgraph/index.html