Parallel Techniques for Compressing and Querying Massive Social Networks

Sudhindra Gopal Krishna, Aditya Narasimhan, Sridhar Radhakrishnan
School of Computer Science
University of Oklahoma
Norman, OK
{sudhi, adinaras, sridhar}@ou.edu

Chandra N Sekharan
Department of Computer Science
Texas A&M Corpus Christi
Corpus Christi, TX
csekharan@tamucc.edu

***Abstract*—The growing popularity of social networks and the massive influx of users have made it challenging to store and process the network/graph data quickly before the properties of the graph change due to graph evolution. Storing graphs or networks that represent entities and their relationships (such as individuals and their friends/followers in a social network) becomes more difficult as the number of users increases, resulting in massive graphs that are challenging to store in standard structures like matrices or adjacency lists. Research in this field has focused on reducing the memory footprint of these large graphs and minimizing the extra memory required for processing. However, there is a trade-off between time and space, as rigorous redundancy removal to achieve a small memory footprint consumes time, and querying becomes more time-consuming when traversing compressed structures compared to matrices or adjacency lists.**

In this paper, we introduce a parallel technique for constructing graphs using compressed sparse rows (CSR), which offers a smaller memory footprint and allows for parallel querying algorithms, such as fetching neighbors or checking edge existence. We extend our work to include parallel time-evolving differential compression of CSR using the prefix sum approach. Additionally, we measure the speed-up gained by using multiprocessors to compress the graph data. To evaluate our techniques, we perform empirical analysis on massive anonymized graphs, including LiveJournal, Pokec, Orkut, and WebNotreDame, which are publicly available. Overall, our results demonstrate that our proposed methods achieve a smaller memory footprint and faster querying compared to traditional storage structures, with additional speed-up gained (up to 83% for the biggest graph with 3.07M nodes and 117.18M edges) through the use of multiprocessors.

***Index Terms*—Massive Social Networks, Compression, Graph Algorithms, Time-Evolving**

I. Introduction

Graphs can represent real-world data from a wide variety of domains. The relationships among the data are captured by the characteristics of the graph. A graph is defined as $G = (V, E)$, where V is a non-empty set of nodes (vertices), and E is a set of relationships (edges). The most common examples of graphs that follow the definition would be a snapshot of a social network, transportation network, biological network, and infrastructure network. Since these graphs change in nature, the word snapshot captures a frame of the network at a particular point in time. Analyzing such graphs/networks offers a wide range of information, starting from how a user's influence would change his connections, the edge betweenness of the highways connecting major cities, analyzing the spread of infection, and designing an efficient routing algorithm based on the nature of the network. Efficiently answering these questions would be quite tedious as one has to deal with the balance between time and space required to process each one of these real-world networks.

In a real-world scenario, these graphs in a matrix format would require massive memory. For example, the Friendster network [1], which contains 65 million nodes and 1.8 billion edges, requires about 30.02 Petabytes of storage space. This kind of storage availability is unheard of, and one way to address this issue is through compression. But once the data are compressed, the data should also be amenable to queries without completely decompressing. So, to come up with a good storage system that queries the graph without decompressing it has been proposed by Boldi and Vigna [2], Nelson et al. [3], [4], Caro et al. [5], Chierichetti et al. [6], Gopal et al. [7].

However, all these compression techniques proposed take time to compress the data. To speed up the process, in this paper, we introduce a parallel algorithm to construct one of the most commonly used graph data structures, Compressed Sparse Row (CSR) [8]. Along with the compression, we also introduce graph querying techniques, where multiple queries can be performed in parallel.

A time-evolving graph is a graph that changes over time and can be represented using a series of graphs at different instances. A graph $G_t = (V_t, E_t)$ where time t indicates an instant that is spread over a certain interval. For instance, the pages on Wikipedia change over time with the addition and deletion of content, and the edited information is saved, allowing the preservation of the page's integrity while remaining open to editing. The information for such pages with time-evolving data can be stored as graphs, which can be useful for various kinds of analysis. However, most real-world graphs are large, and the memory requirements to store the data can be significant. Therefore, the data must be compressed to fit in the main memory for analysis.

The data for time-evolving graphs can be stored in three different representations: adjacency matrix, adjacency list, and edge list. Descriptive, diagnostic, predictive, and prescriptive analyses can be performed on such graphs based on the avail-

979-8-3503-1199-0/23/$31.00 ©2023 IEEE
DOI 10.1109/IPDPSW59300.2023.00140

ability of the data over time. However, with large sizes of real-world graphs such as Wiki-edits and Yahoo Netflow, which are 5.7 GB and 19 GB in edge list format, respectively, storing and analyzing the data can be challenging. Therefore, compressing the data is necessary to store and perform computation on time-evolving graphs.

The contributions are as follows.

- We provide a parallel novel implementation to compress a given edge list into CSR in Section III.
- A parallel prefix sum calculation approach is used to parallelize the construction of a cumulative degree array in Section III-A1.
- The algorithm provided in Section III-A2, computes the degree of each node concurrently.
- Algorithms to parallelly compress a time-evolving graph stored as a CSR are provided in Section IV.
- The neighbor query algorithm that is performed in parallel to get a set of neighbors, given a CSR is explained in section V-A.
- Two approaches to query edge existence(Section V-B):
  - Given an array of edge existence queries, perform subset queries in parallel.
  - Given a single query that parallelly accesses CSR.
- We evaluate (Section VI) the construction of CSR with respect to the number of processors, and experimented with inputs of million-scale social networks.

II. Related Work

The Compressed Sparse Row (CSR) [8] data structure is widely utilized for graph representation. CSR involves compressing each row of the graph into two arrays for each node, allowing efficient packing of all the necessary information into a single array for fast traversal of the data structure. Figure 1 shows the CSR representation of the graph shown in Table I. While CSR has the disadvantage of being a static storage format that can require shifting the entire edge array when adding an edge, its cache-friendliness inspired the development of Packed Compressed Sparse Row ($PCSR$) [9]. $PCSR$ substitutes the edge array in CSR with a Packed Memory Array (PMA) [10], [11], which offers an (amortized) $O(log_2|E|)$ update cost and asymptotically optimal range queries. In this paper, we do not take the packed CSR route to compress the given graph.

Calculation of the prefix sum is one of the crucial steps in the construction of CSR for the calculation of the degree array. The prefix sum operation [12], [13] takes an array A as input of length $n$and outputs an array A' where $\forall i \in \{0, 1, ...n-1\}$,

$$A'[i] = \sum_{j=0}^{i} A[i]$$

There have been parallel in-place algorithms for finding prefix sum [12] that take $O(n)$ work and $O(logn)$ in time. Because of the high dependency on computing parallel degree arrays in CSR, there are many challenges involved. Parallel Packed Compressed Sparse Row ($PPCSR$) [13], designs and analyzes a parallel PMA approach and compares it to other similar approaches [14]–[16].

One way to represent a time-evolving graph is by using a sequence of static graphs, where each graph represents the state of the graph at a specific point in time. These individual graphs, also called snapshots, can be represented as 2D matrices. By stacking these matrices along a third dimension, we can construct a 3D matrix, commonly referred to as a *presence matrix* according to [17].

Caro et al. introduced $ck^d-trees$ in 2016 [5]. They define a contact as a quadruplet (u, v, t_i, t_j) and use this to compress the 4D binary matrix representing the time-evolving graph. This is achieved by treating the 4D matrix as a $k^d tree$ and differentiating between white nodes, which have no contacts, black nodes, which only have contacts, and gray nodes, which have only one contact. This approach is based on the work of Brisaboa et al., who introduced $k^2 - trees$ in 2014 [18].

The G* database [19] is a distributed index that addresses the space issue of the presence matrix by storing new versions of an arc as a log of changes instead. It accomplishes this by storing versions of the vertices as adjacency lists and maintaining pointers to each time-frame. Whenever an arc changes in the next frame, a new adjacency list is created for that vertex's arc, and a pointer is added to the new frame. DeltaGraph [20] is another distributed index that groups the different snapshots in a hierarchical structure based on common arcs.

EveLog [21] is a compressed adjacency log structure based on the "log of events" strategy, consisting of two separated lists per vertex, one for the time-frames and another for the arcs related to the event. The time-frames are compressed using gap encoding, and the arc list is compressed with a statistical model. However, query times suffer because the log must be scanned sequentially. To determine if an arc is active at a particular time-frame in the log strategy, it is necessary to sequentially read the log of events (possibly deactivating/reactivating the arc) until the time-frame is reached. This approach is slow for large time-evolving graphs since it takes linear time. Ferreira et al. [17] follow this strategy by providing a quadruplet $(u, v, t, state)$ for each time an arc changes. In [22], the authors present a data structure of adjacency lists where each neighbor has a sublist indicating the time intervals when the arc is active to improve query times. EdgeLog [21] compresses this idea using gap encoding.

In [23], the FVF (Find-Verify-Fix) framework is developed, which includes a copy+log compression that also supports shortest paths and closeness centrality queries. Three different methods to index time-evolving graphs based on the copy+log strategy are described in [24] and [25].

Two "log of events" strategies, CAS and CET, are proposed in [21] to address the problem of slow query times when processing a log. CAS orders the sequence by vertex and adds a Wavelet Tree [26] data structure to allow for logarithmic time queries. CET orders the sequence by time and develops a modified Wavelet Tree called Interleaved Wavelet Tree to also allow logarithmic time queries.

In 2014, Brisaboa et al. [27] adapt compressed suffix arrays (CSA) [21] for use in temporal graphs (TGCSA) by treating the input sequence as the list of contacts. An alphabet consisting of the source/destination vertices and the starting/ending times is used.

III. Compressed Sparse Row

The compressed sparse row, known as CSR, is one of the most common data structures for storing a graph. CSR was first introduced by Tinney et al. in 1967 [8]. Since then, the structure has been an integral part of research in the area of data storage. The representation consists of three arrays,

- iA: indicates the number of non-zero elements that are present in a row
- jA: indicated the column number where the non-zero element is present in that particular row
- vA: a value array (if the graph is weighted).

If the graph is unweighted, we ignore the third array since an unweighted array is also a boolean array.

TABLE I
An example of a 10-node sparse graph.

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	1	0	0
2	0	0	0	0	0	0	0	1	0	0
3	0	0	0	0	0	0	0	0	1	1
4	0	0	0	0	0	0	0	0	0	1
5	1	0	0	0	0	0	0	0	0	0
6	0	1	0	0	0	0	0	0	0	0
7	0	1	1	0	0	0	0	0	0	0
8	0	0	1	1	0	0	0	0	0	0
9	0	0	0	1	0	0	0	0	0	0

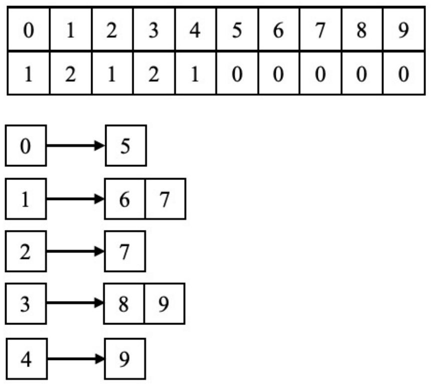


Fig. 1. Compressed Sparse Row representation of the upper triangular matrix of the shown graph as degree array and neighbor list

A. Parallel Construction for CSR

Before we discuss the construction of CSR, we first explain one of the pivotal algorithms used in the construction which is Parallel Prefix Sum by Guy Blelloch [12]. At first, we describe a way to use the prefix to compute the degree array, and later in the paper, we also discuss the concept of prefix sum to construct time-evolving parallel CSR or $TPCSR$.

Algorithm 1: Parallel Prefix Sum calculation

```
Input: An array of unsigned integers $vec$, $startI$,
       $endI$
Output: Prefix sum calculated for $vec[startI : endI]$
1  begin
2      for $i$ = $startI + 1$ to $endI$ do
3          $vec[i]$ += $vec[i-1]$
4      sync() // synchronize all parallel processors
5      // lock to add and carry over the last prefix value
        to each processor
6      Lock()
7          if $startI > 0$ then
8              $vec[end-1]$ += $vec[start-1]$
9      Unlock()
10     sync()
11     if $startI > 0$ then
12         for $i$ = $startI$ to $endI$ do
13             $vec[i]$ += $vec[start-1]$
14     return $vec$
```

1) Prefix Sum Computation: Algorithm 1, of the Parallel Prefix Sum calculation, also known as the Scan algorithm. This algorithm is used to calculate the prefix sum of an input array in parallel, meaning that it can be executed by multiple processors or threads simultaneously.

Given an array of unsigned integers, vec, and two indices $startI$ and $endI$, which define the range of the array to operate on as inputs, this range is referred to as a chunk. The output of the algorithm is the prefix sum calculated for the elements in the specified range. The algorithm starts by iterating over the elements in the input array, starting at index $startI + 1$ and ending at index $endI$. For each element i, the value of $vec[i]$ is updated by adding the value of $vec[i-1]$ to it, thus calculating the prefix sum up to that point, as shown in lines 2-3.

Once the update to the chunk is completed, the algorithm synchronizes all parallel processors. This ensures that all processors have completed their updates and are ready to proceed, lines 4-5. Now, we lock the execution and add the last prefix value to each processor. These updates happen for chunks not starting from 0, as chunk 0 will have no dependency. This is necessary to carry over the prefix sum value from the previous range, lines 6-7. After the process finishes adding the corresponding values, the execution unlocks and synchronizes once again to proceed further to line 8. Finally, all processors except the first pick the previous processor's last element and add it to the range in its chunk of the array. This process is repeated until $endI - 1$ since the end value was updated in the previous step, lines 9-11. The algorithm returns the modified input array, which now contains the prefix sum values for that given range, line 12.

Figure 2 shows the detailed visualization of the work of the

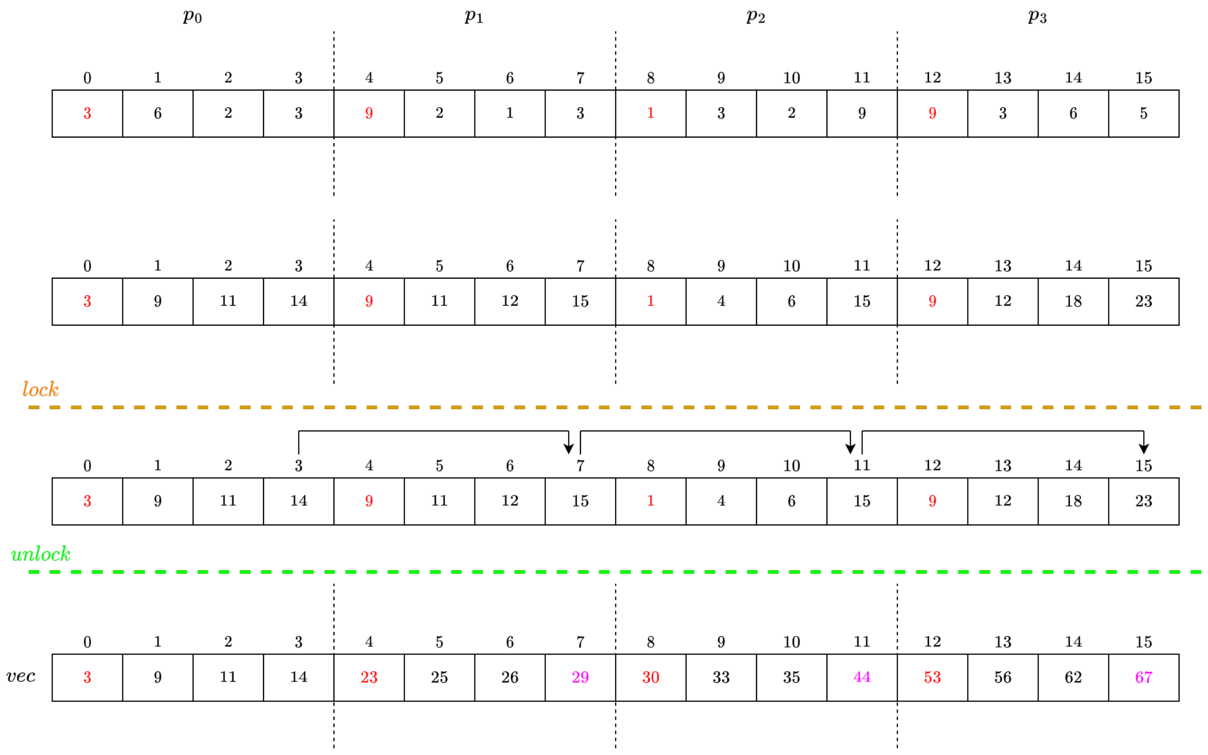


Fig. 2. Shows the steps for obtaining the prefix sum of an array in parallel: The first array is shown in the input, and the dotted lines show the chunks. The second array shows the prefix sum calculated per chunk. The third array, after locking the last element in each chunk is sequentially added to the last element of the next chunk. After unlocking, the processors in parallel add the last element of the previous chunk to all but the last element in the current chunk.

prefix sum.

2) Degree Computation: The degree array (iA) in CSR often stores the starting index of each row. To compute the starting index, one must first compute the degree of each node and then compute the sum. To compute the sum sequentially, in the worst case, one would require $O(n^2)$ units of time, where n is the number of nodes. But to avoid spending $O(n^2)$ units of time, we use the prefix sum algorithm, which is set to solve in $O(logn)$ units of time with $O(n)$ processors.

To compute the degree of our graph, we first split the given array of size n into p chunks, p being the number of processors. Then each processor takes in a chunk of data to compute the occurrences of each element in that particular array, and write it into the $globalDegArray$, which contains the degree of each node. To avoid the concurrency which might occur when two processors are competing to write the degree of same node, we construct a $tempGlobalDegree$ array of size p, then finally merge $globalDegArray$ and $tempGlobalDeg$.

The construction of the degree array is split into two algorithms.

Algorithm 2 computes the degree of a chunk and stores the result in a global array. The algorithm starts by computing the node numbers for the start and end of the chunk. This is done by dividing the array into smaller nodes or subarrays and assigning node numbers to each of them. Then, the variable $uStart$ is assigned the node number for the start of the chunk, and $uEnd$ is assigned the node number for the end of the chunk.

Algorithm 2: Computation of degree array per chunk

```
  Input: An array of unsigned integers $A$, $startI$, $endI$
1 begin
2     $uStart$ = node number at the start of chunk
3     $uEnd$ = node number at the end of chunk
4     $globalTempDegree[A[uStart]]$ = count the
        number of consecutive occurrences of the first
        node in each chunk and store it in a secondary
        global degree array. for $i$ = $startI + 1$ to $endI$ do
5         $nodeI = A[i]$
6         /* denotes looping through each value in $A$ */
7         $globalDegArray[nodeI]$ = count number of
            consecutive occurrences of $nodeI$
```

Next, the algorithm counts the number of consecutive occurrences of the first element in the chunk, which is $A[uStart]$, and stores it in a secondary global degree array called $globalTempDegree$. This is done by looping through the chunk and incrementing a counter variable every time the first element is encountered consecutively.

After that, the algorithm enters a loop that iterates through each element in the specified range, starting from the second element $(A[startI + 1])$. For each element, the algorithm counts the number of consecutive occurrences of

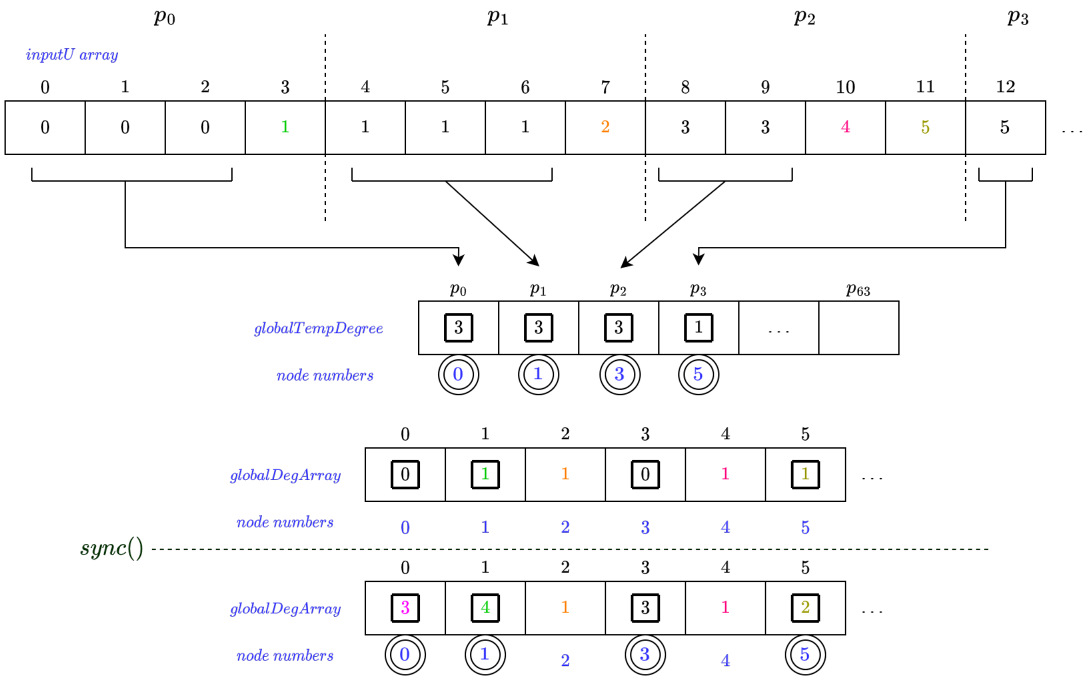


Fig. 3. Shows the working of degree computation in parallel: The first array shown is the input array. The frequency of the node in each chunk is stored in a global temporary degree array. The frequency of the remaining nodes in each chunk is stored in the global degree array. After all, processors finish the degree count for their chunk, the processors are synchronized to ensure that all global degree arrays are up-to-date. Then, we add the frequency of the first appearing node of each chunk to their corresponding degree in the global degree array.

the element and stores it in the global degree array called $globalDegArray$. This is done by looping through the chunk and incrementing a counter variable every time the current element is encountered consecutively.

Finally, the algorithm terminates after looping through all the elements in the specified range and counting the number of consecutive occurrences of each element in the global degree array.

Algorithm 3, builds the degree array for the CSR structure. Figure 3, shows how we merge the $globalDegArray$ with the $globalTempDegree$ array. Since each chunk receives a sorted list of edges, it is for sure that there would only be at most one overlap between two processors.

For example, in the figure, the chunk 1 has received edges from nodes $0-1$, the chunk 2 has received edges from nodes $1-2$, the chunk 3 has received edges from nodes $3-5$, and finally, chunk 4, has element 5. Each processor in the algorithm will always save the frequency of the first element in the temporary global degree array, and the remaining elements are written directly to the global degree array. Once all processors have finished, they synchronize their computations using the **sync()** function to ensure that all the global degree arrays are up-to-date.

After that, each processor updates the global degree array $globalDegArray$ by adding the temporary degree count $globalTempDegree[pid]$ for the first element in its chunk, which is calculated as $A[pid * chunkSize]$. This is done to account for the overlap between consecutive chunks in the CSR format.

Finally, the algorithm returns the global degree array $globalDegArray$, which is needed to represent the degree of each node in the graph in CSR format.

Algorithm 3: Build CSR degree array

```
Input: An array of unsigned integers $A$, $p$ number of processors
Output: Degree array $degArr$
1 begin
2     $globalDegArray$ // an array of size $n$
3     $globalTempDegree$ // an array of size $p$
4     $chunkSize = n/p$
5     do in parallel:
6         call Algorithm 2 for each chunk
7         sync()
8         $globalDegreeArray[A[pid * chunkSize]] += globalTempDegree[pid]$
9     return $globalDegreeArray$
```

3) Build CSR: Once we retrieve the degrees of all nodes in the graph, these graphs are now compressed. For further compression, we are using our novel technique to store the integer numbers associated with both the degree array iA, and the edge column array jA.

Similarly to our previous approach of diving the array into chunks and providing them to each processor, we continue the

Algorithm 4: Build bitPacked CSR

```
Input: EdgeList, $p$ number of processors
Output: BitPacked $CSR$
1 begin
2     do in parallel: for each processor
3         $chunkSize$ = Compute the chunk based on the processor availability.
4         call bitPack algorithm from [7], for each chunk.
5         The resultant bit array is then stored in a global location.
6     $finalBitArray$ = merge all bitArrays from global location
7     Repeat the process once for degree array $iA$,
8     and once for edge column array $jA$ return $CSR$
```

same approach to call our bit packing algorithm mentioned in [7], to compress the CSR. We repeat this process separately for the degree array and again for the edge column array, as shown in Algorithm 4.

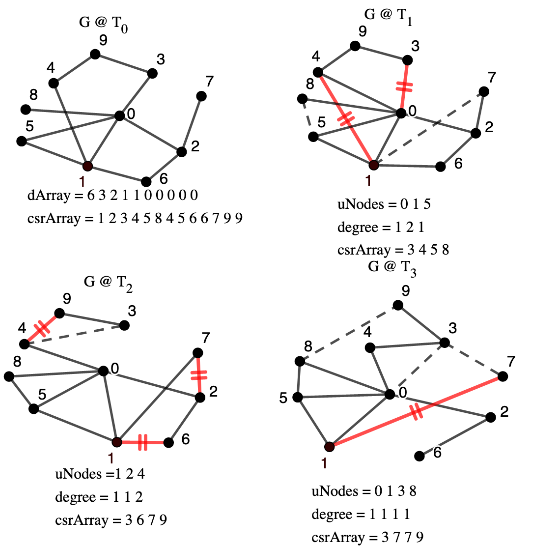
IV. Parallel Construction of Time-Evolving CSR

For every input of the time-evolving graphs G, the input is divided as an ordered triplet (u, v, T_τ), where u and v are the nodes that form an edge at time T_τ. If the edge appears again later in another time-frame $T_{\tau+i}$, the edge is considered to be deactivated in the time-frame. We assume that the datasets are sorted with respect to the time-frames and then sorted by node numbers for each time-frame.

Figure 4 shows the design of how storing a differential time-evolving graph works. The graph shown in the figure evolves for 4 time-frames in every time-frame, we could either see an edge being added or an edge being deleted or no change. To illustrate, we have shown the edge being deleted in red color and the edge being added as a dotted line. For time-frame T_0 (first time-frame), we construct the CSR with both the degree array and the column index array, and for the following time-frames, T_i, we store the difference in the graph with respect to the time-frame $T_{(i-1)}$.

Since the process is serial and has dependency over the previous time-frame, constructing a time-evolving graph in parallel would need a different approach. Figure 5, shows the working on parallelly constructing time-evolving CSR. The input to the compression method is a time-sorted edge list. Therefore, we now divide the entire edge list. Once divided, we compute CSR on these chunks. Note that there could be an overlap similar to that of computation of degree in Section III-A2. Similarly to degree merging, we merge the overlapped CSR to obtain one CSR for every time-frame.

Storing the CSR this way is space-consuming, as not all nodes have changed state from one-time-frame to another. Therefore, in the next step, we perform a differential operation parallelly. To perform this differential operation, we seek to prefix the sum computation. Now, we divide the array of



CSR as unsinged char: 01101100010010001000 1000010011000010101000100101011100001001100001010 10 000100101 100010100 110000101010001 100010001 100100010 0010111000010101000 0000100011000001 1111 1100111011101001

Fig. 4. Captures the graph evolving over 4 time-frames. The edges in red show the edge being deleted from one time-frame to another, and the dotted edge indicates the edge being added.

$CSR's$ into chunks and process the differences. The first time-frame in every chunk is kept as is, and the differences are computed on the remaining $CSR's$ in the same chunk.

Once we find the differences in each chunk, we perform the sync and lock operation to propagate the end difference. Once the end differences are propagated, we unlock and resync to perform the final differential operation, similar to prefix sum.

The differences here are the edges added or deleted in each time-frame. Within a given time interval, if an edge appears an even number of times, the edge is set to be inactive, and if the count is odd, then the edge is set to be active. The working of $TCSR$ is shown by Gopal et al. [7].

Algorithm 5: Build $TCSR$ degree array

```
Input: EdgeList with time-intervals, $p$ number of processors
Output: Degree Array $degArrT$ as differential $TCSR$
1 begin
2     do in parallel:
3         Divide the input edge list, and construct $CSR$ for each time-frame in the chunk.
4         Merge overflowing $CSR's$ between chunks
5         Perform differential $CSR$ for every time-frame using the prefix sum algorithm.
6     return BitArray $TCSR$
```

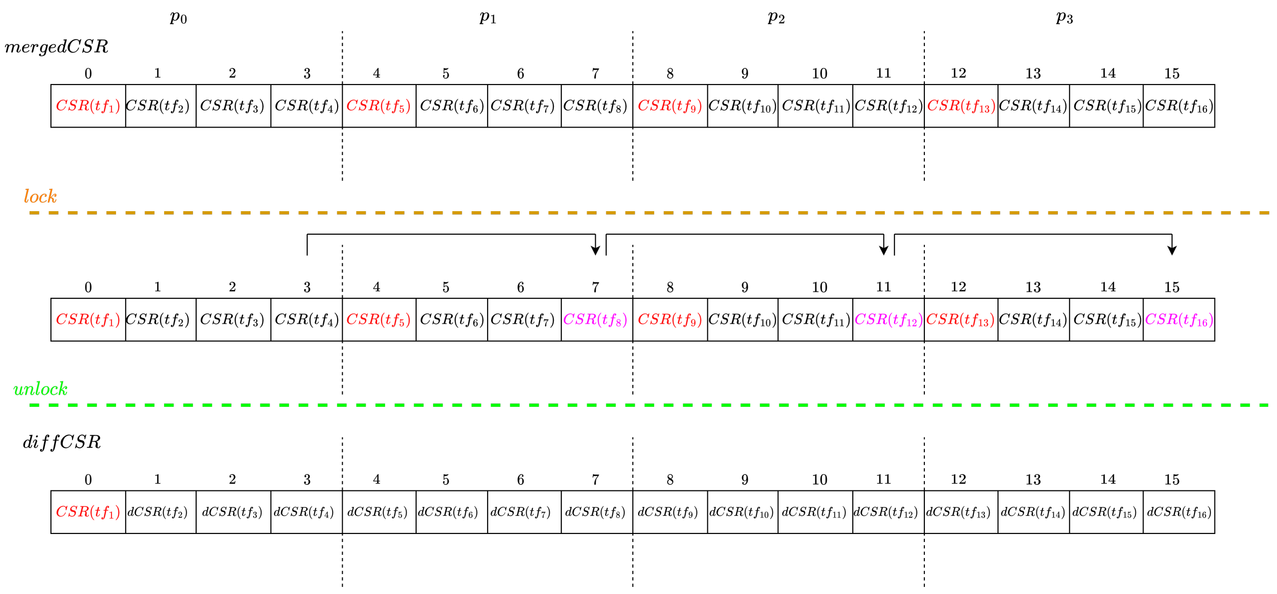


Fig. 5. Shows the construction of time-evolving differential CSR using Prefix Sum, which is used to compute the difference in the consecutive time-frames. This follows a similar approach to Figure 2.

V. Parallel Querying Algorithms for CSR

One of the most important operations on a social network is about getting to know if there is a connection between two individuals or checking who are all the acquaintances of a given user. These operations translate to checking if there is an edge between two nodes and fetching all neighbors of a given node.

These two searches are performed quite frequently, which means that we can search for multiple queries at once. For a social network with millions and billions of users that are using it at once, it is quite time-consuming to perform one query at a time. Instead, if multiple processors involve querying multiple items at once, the time required to search reduces.

In this paper, we propose two querying algorithms, one is to perform an array of neighborhood queries in parallel, and the second one is to perform an array of edge existence queries in parallel. In addition, we propose a quicker way to query the edge existence by splitting the bit array of a node into multiple chunks and making multiple processors check for the edge.

A. Neighborhood Query

The first querying algorithm we perform is neighborhood querying. Given an array of queries (node numbers $uNodes$), an array of unsigned bits A, the start and end indices in $uNodes$, and the number of bits $numBits$. The $numBits$ tells us the number of bits to process in the bit array to obtain a node number.

The algorithm starts by iterating through the range of nodes in $uNodes$ from the specified start index $startI$ to the end index $endI$. For each node $uNodes[i]$ in this range, the algorithm calls the $GetRowFromCSR$ function mentioned in [28], passing in the array of unsigned bits A, the starting index of the node $uNodes[i].startingIndex$, the degree of the node $degrees[uNodes[i]]$, and the number of bits $numBits$.

The $GetRowFromCSR$ function takes as input a compressed sparse row (CSR) representation of the graph and returns the row corresponding to the specified node. This row contains the indices of the node's neighbors in the graph.

The row of neighbors returned by $GetRowFromCSR$ is then assigned to the corresponding element in the resultNeighbors vector, using the node number as the index.

Algorithm 6: Get neighbors of $uNodes$

```
Input: An array of unsigned bits $A$, an array of
       $uNodes$ containing neighbor queries, $startI$
       index in $uNodes$, $endI$ index in $uNodes$, and
       the number of bits $numBits$
Output: Vector of vectors $resultNeighbors$ of all the
        neighbors of $uNodes$ from $startI$ to $endI$
1 begin
2     for $i$ = $startI$ to $endI$ do
3         /* denotes looping through one neighbor from
            $uNodes$ */
4         $resultNeighbors[uNodes[i]]$ =
            $GetRowFromCSR(A,$
            $uNodes[i].startingIndex,$
            $degrees[uNodes[i]], numBits)$
```

Finally, once the loop ends and the queries are completed, the vector $resultantNeighbors$ will be left with an array of neighbors. During the call to this method, the array of queries is split among processors, and the chunks of neighbors are obtained at once.

B. Edge Existence

In this section, we are going to discuss two types of edge existence; the first one being, given an array of edges, query the existence, and the second one being, given an edge, divide the array into chunks and process for existence.

Algorithm 7, takes an array of unsigned bits A, an array of edges $edges$ containing edge queries, $startI$ index in $edges$, $endI$ index in $edges$, and the number of bits $numBits$ as input, and outputs the existence of edges between all pairs of nodes u and v in $edges$.

The algorithm iterates through each edge query in the given range of indices from $startI$ to $endI$. For each edge query, it first retrieves the neighbors of the source node u in the given array A using the function $GetRowFromCSR$ function mentioned in [28] which takes A, the starting index of the row of the source node, the degree of the source node u and the number of bits $numBits$ as input and returns the list of neighbors of the source node.

Next, the algorithm iterates through each neighbor of the source node u and checks if it is equal to the target node v. If a match is found, the algorithm outputs the presence of an edge between the nodes u and v.

Algorithm 7: Edge existence of an array of edges

```
Input: An array of unsigned bits $A$, an array of $edges$
       containing edge queries, $startI$ index in $edges$,
       $endI$ index in $edges$, and the number of bits
       $numBits$
Output: Existence of edge between $u$ and $v$ for all
        $edges$
1 begin
2   for $i$ = $startI$ to $endI$ do
3     $uNeighs$ = $GetRowFromCSR$($A$,
        $edges[i].startingIndex$, $degrees[edges[i]]$,
        $numBits$)
4     for $s_n$ in $uNeighs$ do
5       /* denotes looping through each neighbor
          of $u$ */
6       if $s_n == v$ then
7         output presence of edge $(u, v)$
```

Overall, the algorithm checks for the existence of edges between all pairs of nodes u and v in the given range of edge queries. It does this by retrieving the neighbors of the source node for each edge query and then checking if the target node is present in the list of neighbors.

For the algorithm 8, we narrow down the search space for an edge query by first retrieving the neighbor of u and splitting the neighbor array among processors to search for v. This could also be extended to a binary search to speed up the process.

So far, we have seen how each of the querying algorithms works. However, the work lies in the design of the call to algorithms parallelly. Algorithm 9 shows the call to all querying algorithms parallelly.

Algorithm 8: Single Edge Existence

```
Input: An array of neighbors of $uNeighs$, $startI$ in
       $A$, $endI$ in $A$, node $v$
Output: Existence of $v$ in $uNeighs$
1 begin
2   for $s_n$ in $uNeighs$ within the range do
3     /* denotes looping through each neighbor of $u$
        */
4     if $s_n == v$ then
5       output presence of edge $(u, v)$
```

The first parallel do, explains the call to Algorithm 6, the algorithm fetches all neighbors associated with the input array of nodes by splitting the input array into p parts. Once the input is split, each processor takes in the compressed CSR and the start and end of the query array to fetch all the neighbors. The result for every node queried will be returned as an array of arrays with all the neighborhood information.

The second parallel do, explain the call to Algorithm 7, where given an array of edge inputs, how do we parallelly see if the edge exists or not? The approach is similar to a neighborhood query, where we divide the array into p chunks and divide the input amongst multiple processors. Each processor then reads the input and processes the adjacency list associated with the edge to see if the edge exists.

The third parallel do, explain the call to Algorithm 8, where given a single edge (u, v), query to check if the edge exists in parallel. For this, we first retrieve the neighborhood list of the node u, then split the list into p parts, and parallelly subject each processor to look if v exists in the given chunk.

Overall, this algorithm is designed to efficiently query a compressed CSR data structure in parallel by dividing the work among multiple processors. By doing this, the algorithm can speed up the process of querying large matrices or graphs.

VI. Experimental Evaluation

In this section, we evaluate the performance of parallel CSR. For evaluation, we have considered publicly available social networks provided by Stanford SNAP [1].

Table II shows the compression result on various numbers of processors. The first three columns explain the properties of the graphs, the number of nodes, and the number of edges present in each graph. The fourth column shows the space required to store the graph if the graph is stored in an edge list. The size might seem small compared to storing the graph in a matrix. However, the edge list consumes more time in querying compared to CSR. Therefore, in the fifth column, we have the space required to store the same graph in bit packed CSR. The sixth column shows the different numbers of processors on which the graph was tested; if the number of processors is equal to 1, then the algorithm is said to run

Algorithm 9: Call to Querying Algorithms

```
Input: Compressed $CSR$
1  begin
2      // Given an array of nodes, and $p$ processors,
3      // fetch all neighbors associated with these nodes.
4      do in parallel:
5          Split the input array into $p$ parts, and call
             Algorithm 6
6          This algorithm takes in the $CSR$, and the start
             and end of the query array for each processor
7          The result for every node queried will be
             returned as an array of arrays with all the
             neighborhood information.
8      // Compute edge-existence given multiple edges
9      do in parallel:
10         Give an array of edges to be queried,
11         Split the edge array into to $p$ parts,
12         Call Algorithm 7, with the query array and
             array starting and ending index.
13     // Given the list of adjacency for a node $u$, and $p$
         processors
14     // To see if $v$ exists.
15     do in parallel:
16         Split the list into $p$ parts, and call Algorithm 8
17         This algorithm takes in the starting and ending
             index of the list along with the node to be
             searched.
18         One of the processors will return true if the
             edge exists,
19         If not all return false
```

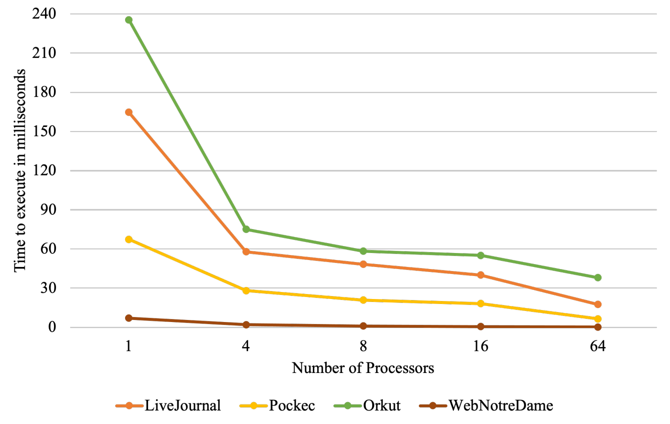


Fig. 6. Shows the execution times for different number processors for various graphs. We see the time taken to construct CSR decreases significantly when parallelized.

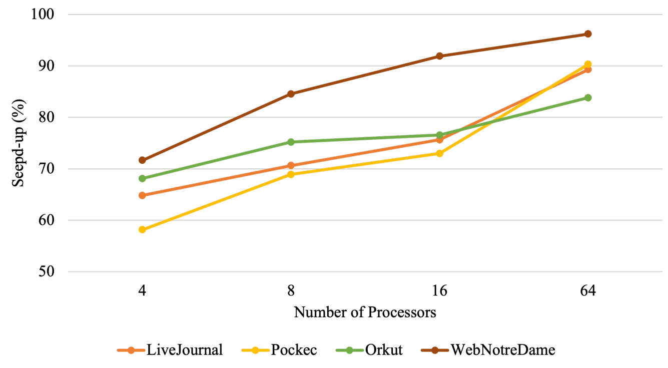


Fig. 7. Shows the speed-up gained using multiple processors to compress the graphs to CSR.

in serial mode. Therefore, the seventh column shows the time required to process the same graph when there are a different number of processors working to achieve the resultant CSR. The final column shows the speed-up gained using multi-processors over a single processor, measured in percentage (%), as shown in figure 7.

Figure 6 shows the pictorial representation of the time taken to compress the graph versus the number of processors used to compress it.

A rapid decline is seen when going from 1 processor to 4, then a steady decline with 8 and 16, followed by a decent drop in time with 64 processors. The steady decline within multiple processors is due to the inherent sequential steps.

All experiments were run on AMD Ryzen Threadripper 3970X 32-Core Processor, with 128 GB of memory, and the programs are written in GNU C++17.

VII. Conclusion

In conclusion, the analysis of social networks can provide valuable insights, but as the size of these networks grows, storing them for analysis becomes a challenge. Although various storing mechanisms are available, compressing a graph for storage takes time, and accessing the information directly from a compressed structure is not always straightforward.

In this paper, we solve the problem by speeding up the compression process on CSR and also increasing the number of queries that can be performed at once. To aid in the process of constructing the CSR in parallel, we provide a parallel prefix sum approach to compute the degree array concurrently. We also propose algorithms to construct the time-evolving differential CSR in parallel using prefix sum.

Overall, the contributions of this paper provide a valuable foundation for efficient parallel graph processing, which is essential for dealing with the increasingly large and complex graphs that arise in many real-world applications.

References

[1] Stanford Network Analysis Project, “Stanford Large Network Data Collection.” https://snap.stanford.edu/data/index.html, 2011.

[2] P. Boldi and S. Vigna, “The Webgraph Framework I: Compression Techniques,” in *Proceedings of the 13th International Conference on World Wide Web*, WWW ’04, (New York, NY, USA), pp. 595–602, ACM, 2004.

[3] M. Nelson, S. Radhakrishnan, A. Chatterjee, and C. Sekharan, “Queryable Compression on Streaming Social Networks,” in *Big Data (Big Data), 2017 IEEE International Conference on*, IEEE BigData ’17, IEEE Computer Society, 2017.

TABLE II
SHOWS THE EXPERIMENTS PERFORMED ON THE DIFFERENT TYPES OF GRAPHS.

Graphs	# of Nodes	# of Edges	EdgeList Size	CSR	# of Processors	Time (ms)	Speed-Up (%)
LiveJournal	4,847,571	68,993,773	1.1 GB	24.73 MB	1	164.76	-
					4	57.94	64.83
					8	48.35	70.65
					16	40.09	75.67
					64	17.613	89.31
Pockec	1,632,803	30,622,564	405 MB	197.83 MB	1	67.41	-
					4	28.19	58.18
					8	20.95	68.92
					16	18.21	72.99
					64	6.53	90.31
Orkut	3,072,627	117,185,083	1.7 GB	313.19 MB	1	235.52	-
					4	75.09	68.12
					8	58.38	75.21
					16	55.15	76.58
					64	38.09	83.83
WebNotreDame	325,729	1,497,134	22 MB	3.82 MB	1	7.13	-
					4	2.02	71.67
					8	1.1	84.57
					16	0.577	91.91
					64	0.27	96.21

[4] M. Nelson, S. Radhakrishnan, and C. Sekharan, "Queryable Compression on Time-Evolving Social Networks with Streaming," in *Big Data (Big Data), 2018 IEEE International Conference on*, IEEE BigData '18, IEEE Computer Society, 2018.

[5] D. Caro, M. A. Rodriguez, N. R. Brisaboa, and A. Farina, "Compressed kd-tree for temporal graphs," *Knowl. Inf. Syst.*, vol. 49, pp. 553–595, Nov. 2016.

[6] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan, "On compressing social networks," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, (New York, NY, USA), p. 219–228, Association for Computing Machinery, 2009.

[7] S. Gopal Krishna, M. Nelson, S. Radhakrishnan, A. Chatterjee, and C. Sekharan, "On Compressing Time-Evolving Networks," in *ALLDATA 2021, The Seventh International Conference on Big Data, Small Data, Linked Data and Open Data*, pp. 43–48, 2021.

[8] W. F. Tinney and J. W. Walker, "Direct solutions of sparse network equations by optimally ordered triangular factorization," *Proceedings of the IEEE*, vol. 55, no. 11, pp. 1801–1809, 1967.

[9] M. Winter, R. Zayer, and M. Steinberger, "Autonomous, independent management of dynamic graphs on gpus," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, IEEE, 2017.

[10] A. Itai, A. G. Konheim, and M. Rodeh, "A sparse table implementation of priority queues," in *Automata, Languages and Programming: Eighth Colloquium Acre (Akko), Israel July 13–17, 1981 8*, pp. 417–431, Springer, 1981.

[11] M. A. Bender, E. D. Demaine, and M. Farach-Colton, "Cache-oblivious b-trees," in *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pp. 399–409, IEEE, 2000.

[12] G. E. Blelloch, "Prefix sums and their applications," 1990.

[13] B. Wheatman and H. Xu, "A parallel packed memory array to store dynamic graphs," in *2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, pp. 31–45, SIAM, 2021.

[14] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 135–146, 2013.

[15] L. Dhulipala, G. E. Blelloch, and J. Shun, "Low-latency graph streaming using compressed purely-functional trees," in *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation*, pp. 918–934, 2019.

[16] J. Shun, L. Dhulipala, and G. E. Blelloch, "Smaller and faster: Parallel processing of compressed graphs with ligra+," in *2015 Data Compression Conference*, pp. 403–412, IEEE, 2015.

[17] A. Ferreira and L. Viennot, "A Note on Models, Algorithms, and Data Structures for Dynamic Communication Networks," Research Report RR-4403, INRIA, 2002.

[18] N. R. Brisaboa, S. Ladra, and G. Navarro, "Compact representation of web graphs with extended functionality," *Information Systems*, vol. 39, pp. 152–174, 2014.

[19] A. G. Labouseur, J. Birnbaum, P. W. Olsen, Jr., S. R. Spillane, J. Vijayan, J.-H. Hwang, and W.-S. Han, "The G* Graph Database: Efficiently Managing Large Distributed Dynamic Graphs," *Distrib. Parallel Databases*, vol. 33, pp. 479–514, Dec. 2015.

[20] U. Khurana and A. Deshpande, "Efficient snapshot retrieval over historical graph data," *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pp. 997–1008, 2013.

[21] D. Caro, M. Andrea Rodríguez, and N. R. Brisaboa, "Data structures for temporal graphs based on compact sequence representations," *Inf. Syst.*, vol. 51, pp. 1–26, July 2015.

[22] B.-M. Bui-Xuan, A. Ferreira, and A. Jarry, "Computing shortest, fastest, and foremost journeys in dynamic networks," Tech. Rep. RR-4589, INRIA, Oct. 2002.

[23] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng, "On querying historical evolving graph sequences," *PVLDB*, vol. 4, pp. 726–737, 2011.

[24] G. D. Bernardo, N. R. Brisaboa, D. Caro, and M. A. Rodríguez, "Compact data structures for temporal graphs," in *2013 Data Compression Conference*, pp. 477–477, March 2013.

[25] S. Álvarez-García, N. R. Brisaboa, G. d. Bernardo, and G. Navarro, "Interleaved K2-Tree: Indexing and Navigating Ternary Relations," in *2014 Data Compression Conference*, pp. 342–351, March 2014.

[26] R. Grossi, A. Gupta, and J. S. Vitter, "High-order Entropy-compressed Text Indexes," in *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, (Philadelphia, PA, USA), pp. 841–850, Society for Industrial and Applied Mathematics, 2003.

[27] N. R. Brisaboa, D. Caro, A. Fariña, and M. A. Rodríguez, "A compressed suffix-array strategy for temporal-graph indexing," in *SPIRE*, 2014.

[28] S. G. Krishna, A. Narasimhan, S. Radhakrishnan, and R. Veras, "On large-scale matrix-matrix multiplication on compressed structures," in *2021 IEEE International Conference on Big Data (Big Data)*, pp. 2976–2985, 2021.