Postmortem Computation of Pagerank on Temporal Graphs

Md Maruf Hossain
mhossa10@uncc.edu
University of North Carolina at Charlotte
Charlotte, North Carolina, USA

Erik Saule
esaule@uncc.edu
University of North Carolina at Charlotte
Charlotte, North Carolina, USA

ABSTRACT

Temporal graphs capture changes in relational data over time and have been of increasing interest to data analysts. Most research focuses on *streaming* algorithms that incrementally update an analysis to account for the changes in the graph. However, one can also be interested in understanding the nature of changes in the graph over time. In such a case, they perform a *postmortem* analysis on different points in time where all the data known in advance

We study in this paper a *postmortem* analysis of *Pagerank* overtime on graphs that are defined by temporal relational event databases. A relation between two entities at a particular point in time will form an edge between these two entities and that will remain in the graph for a fixed period of time.

While one can reuse a streaming algorithm for that purpose, leveraging the availability of all the data from the beginning can be beneficial. Postmortem analysis enables encoding the temporal graph with a more efficient graph representation. Also, it provides an additional level of parallelism since one can not only parallelize within a particular timestamp but also across different timestamps. We will show that depending on the properties of the temporal data, either parallelization can be better, and in some cases, a combination of both approaches is preferable.

We experimentally show across 7 databases and across different temporal derivations of the graph that postmortem analysis can be between 50 times and 880 times faster than streaming analysis.

KEYWORDS

Temporal Graph, Pagerank, SpMM, SpMV, Streaming Graph Analysis

ACM Reference Format:
Md Maruf Hossain and Erik Saule. 2022. Postmortem Computation of Pagerank on Temporal Graphs. In *51st International Conference on Parallel Processing (ICPP '22), August 29-September 1, 2022, Bordeaux, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3545008.3545055

1 INTRODUCTION

Graphs have been used to model various natural, social, and constructed objects and phenomena such as the brain, friendship relations, and the physical road infrastructures. Such models help understanding more deeply the objects we study. They have been used to identify terrorists [5, 7, 28], understand the link between traffic and economic activity [1, 8, 15], or identify keywords in text [12, 32]. There are numerous analyses conducted on these graphs for a different type of usage, including Pagerank [30], betweenness and closeness centrality [3, 16], modularity-optimizing community detection [6, 24], k-core decomposition [14, 34].

These graphs are often analyzed as static graphs, but fundamentally the objects they model evolve over time: Roads are constructed and blocked off; Humans form new relations while others fade away. A more accurate model would be to define a *temporal graph* [22] that has vertices and edges that only exist for some periods of time. A common type of analysis on these temporal graphs is *streaming analysis* where an analysis is performed on the most up-to-date version of the graph. Obviously recomputing the analysis from scratch would be expensive and in many cases, it is possible to perform an incremental update on the analysis by starting from the results of recent analyses and accounting for only the latest changes in the graph. This has been done on many analyses including streaming Pagerank [11, 31], streaming Closeness Centrality and Betweenness Centrality [20, 35], streaming k-core [14, 34], and many others.

We are interested in this paper in a different form of analysis that sees the graph as a time series. In this analysis, we assume that we know the entire temporal graph at the beginning of the analysis; we refer to the analysis as being *postmortem.* (Some people may refer to that sort of analysis as being offline; but we chose not to refer to it this way to avoid confusions.) This is in contrast with a streaming analysis which discovers the graph during the analysis. Various problems on temporal graph have been investigated, including diameter change [26], and rank of web pages change [38] on the web.

We assume that the analysis is conducted at regular interval in time. Also the temporal graphs are defined by edges that appear at a particular point in time and remain in the graph for a constant amount of time. As such, the temporal graph can model edge addition and deletion, as well as vertex addition and deletion.

We will also restrict our analysis to computing Pagerank [30]. It is a simple analysis that is well understood, with known streaming algorithms [11, 31]. And it applies to a wide variety of applications.

In this paper, we show how to perform a postmortem temporal analysis of a graph using Pagerank on a shared-memory parallel system. We show that postmortem analysis is much faster than an equivalent streaming analysis and static (offline) analysis. In particular, we show that postmortem analysis provides benefits over static and streaming execution model. The challenges and contributions of this paper include:

Data Representation: Streaming and static have a fairly well set representation that have their own pros and cons. But in a postmortem analysis, representing the temporal data offers tradeoff between volume of memory and performance of the analysis. We present our data representation in Section 4.1. We investigate and evaluate the tradeoffs.

Leveraging incremental methods: There are several methods to reduce the amount of work when computing Pagerank in a streaming mode. Upon some update, the graph is still quite the same as it was, the values of Pagerank are going to be related, and incremental methods have been developed for Pagerank. Based on existing methods (described in Section 3.3.2), we develop an incremental method appropriate for this particular use case in Section 4.2.

Different Level of Parallelization: In Postmortem analysis, one can compute Pagerank on each graph simultaneously. Of course, the calculation of Pagerank on a particular graph is also a fundamentally parallel computation. Questions of load balance, incompatibility with incremental optimization, and scheduling need to address to benefit the most from modern platforms. We investigate these questions in Section 4.3.

SpMV-style vs SpMM-inspired Postmortem Pagerank: Pagerank is fundamentally similar to a sparse matrix-vector multiplication (SpMV) operation. However, we know that sparse matrix-matrix multiplication (SpMM) can obtain higher performance. We discuss how we take inspiration from SpMM and rephrase the calculation of Pagerank on a temporal graph to obtain the benefits of an SpMM formulation without compromising other optimizations in Section 4.4.

Demonstrate the efficiency of Postmortem analysis: It makes intuitive sense that postmortem analysis offers more avenues for optimization than both offline and streaming analysis. But to what extent is postmortem preferable. We evaluate experimentally the question in Section 6 and show that in our benchmark postmortem analysis can be between 50 times to 400 times faster than streaming analysis.

2 PROBLEM STATEMENT

2.1 Temporal Graph from Temporal Events

Temporal Edge Set: We assume our input is a set of edges of the form $Events = \langle u, v, t\rangle$, where u, v are vertices from some vertex set V (the elements of V known because of offline behavior), and t is an integer timestamp. Without loss of generality, we can assume that entries are listed in increasing timestamp order. We call the entire sequence of such triples a temporal edge set, and each triple is an event.

Note that a streaming model assumes that the elements of the set are disclosed, monotonously in time, over the execution of the application. But in a postmortem model, all the temporal edges are known at the beginning of the application.

Sliding Window Model: We define $G(T_s, T_e)$ as the graph induced by the events that occured between T_s and T_e. That is to say, $G(T_s, T_e) = (V, E)$ where $\{e = (u, v) \in E | \exists (u, v, t) \in Event, T_s \leq t \leq T_e\}$.

In this paper, we are interested in analyzing the sequence of graph $(G_0 = G(T_0,\ T_0 + \delta), G_1 = G(T_1,\ T_1 + \delta), G_2 = G(T_2,\ T_2 + \delta), \ldots, G_m = G(T_m,\ T_m + \delta))$ with $T_i = T_{i-1} + sw\}$ and T_0 is set by the beginning of the dataset. In other words, the temporal graph is defined by a sequence of graphs generated by sliding a window over the time period. The window is of fixed size δ and each window slide by a sliding offset of sw time-units compared to the previous one. This sliding window model is illustrated in Figure 1.

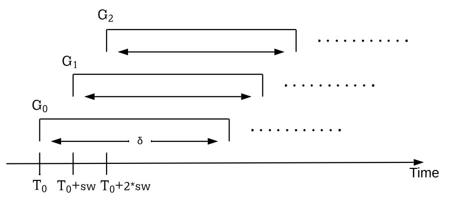
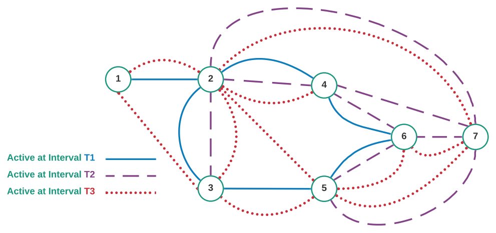


Figure 1: Sliding Window Model

Edges		Edge Arrival Time	Time Interval		
v_1	v_2		T1	T2	T3
1	2	06/21/2021	✓	✗	✗
3	5	06/25/2021	✓	✗	✗
4	6	07/11/2021	✓	✓	✗
2	3	08/01/2021	✓	✓	✓
2	4	08/11/2021	✓	✓	✓
5	6	09/13/2021	✓	✓	✓
2	7	10/02/2021	✗	✓	✓
4	7	10/05/2021	✗	✓	✓
5	7	10/06/2021	✗	✓	✓
6	7	10/09/2021	✗	✓	✓
1	2	11/05/2021	✗	✗	✓
1	3	11/06/2021	✗	✗	✓
2	5	11/09/2021	✗	✗	✓
3	5	11/12/2021	✗	✗	✓

(a) Temporal edge list[Time interval T1 = (6/1/2021-9/15/2021), T2 = (7/1/2021-10/15/2021) and T3 = (8/1/2021-1/15/2022)]



(b) Temporal Graph

Figure 2: Edgelist and temporal graph.

Figure 2a presents an example of a list of temporal edges for a graph. The edges arrive between 06/21/2021 and 11/12/2021. Maybe the analyst is interested in analyzing phenomena that take some time to unfold and select a window of size $\delta = 3\frac{1}{2}$ months. The first graph G_0 includes edges arriving after 6/1/2021 and until 9/15/2021. After that it will move forward the starting time of the second graph G_1 by $sw = 1$ month and the time interval for G_1 will be 7/1/2021-10/15/2021). Figure 2b shows the active edges for the first 3 graphs of the sequence of the temporal graph.

Algorithm 1 Pagerank on Temporal Graph

```
Input: $Events, sw, \delta, T_0, m$
1: $i \leftarrow 0$
2: while $i \leq m$ do
3:     $PAGERANK_i \leftarrow$ PagerankAlgorithm($G(T_i,\ T_i + \delta)$)
4:     $i \leftarrow i + 1$
5:     $T_i \leftarrow T_{i-1} + sw$
6: end while
```

2.2 Postmortem Graph Analysis for Pagerank

Pagerank is a metric of the importance of vertices in a graph, originally used on webpages modeled as a directed graph [30]. Let v be a vertex, $\Gamma^+(v)$ be the set of vertices v points to, and $\Gamma^-(v)$ be the set of vertices that point to v. For a teleportation probability α, the Pagerank(PR) [30] equation for v is recursively defined as:

$$PR(v) = \frac{\alpha}{|V|} + (1-\alpha) \sum_{u \in \Gamma^-(v)} \frac{PR(u)}{|\Gamma^+(u)|} \tag{1}$$

While Pagerank values for each node of the graph could be obtained by solving the system of equations, it is more common to compute Pagerank iteratively. The Pagerank equation is evaluated from previous values of Pagerank. This involves performing one Sparse Matrix-Vector multiplication (SpMV). After some iterations, the values converge to the solution of the equation. Implementations usually numerically check for convergence after each iteration and execute a fixed number of iterations at most. Beamer and Scott *et al.* [4] presented how to reduce Pagerank communication via propagation blocking; and although this paper does not leverage that particular technique, we believe it is compatible.

The problem we are trying to solve is to compute Pagerank on all graphs in the sequence. Sequentially one could solve the problem with the simple method given in Algorithm 1. But one does not have to compute the different Pagerank vectors in-order. They could compute in different orders. Of course, applications will have a downstream analysis that will depend on these vectors.

3 BACKGROUND AND RELATED WORKS

3.1 Applications of the Sliding Window Model

The formulation of the temporal graph based on sliding windows from an event database is appropriate for many applications. Parameters *delta* and *sw* are application parameter. They enable the analyst to explore a dataset at different time scales and resolutions.

For instance, consider the analysis of academic collaboration networks. One can define events based on papers, if authors a_1 and a_2 co-wrote a paper on day d, you insert a tuple (a_1, a_2, d) in $Events$.

Setting a larger value of $\delta = 10$ years will enable the analyst to think of the important of authors in a scientific era. Meanwhile, setting a smaller value of $\delta = 1$ year will enable to study current collaborator dynamic. Neither value for the parameter is inherently better, but they enable to study different social phenomenon. The sw parameter is essentially a resolution parameter. It enables to provide fewer or more points in the generated time series.

The temporal graph constructed this way could be analyzed in various way. While we focus on Pagerank in this paper, different analysis could be done using other kernels like closeness and betweenness centrality, connecting component, k-core, etc.

3.2 Temporal Graph Analysis

We are not the first to analyze graphs temporally from event data. Hossain, Murshed *et al.* analyzed communication network dynamics during organizational crisis [23]. They showed that some actors of an organization that are prominent or more active will become central during the organizational crisis. Now, analyzing this kind of problem requires insight into periodic changes in the dynamic communication graph. Time interval-wise analyses show the impact of actor's changes on the organization and one can find how the role of an actor evolves during a crisis and understand the underlying cause.

Stolman and Matulef [37] proposed a *HyperHeadTail* streaming algorithm which can estimate the degree distribution of a dynamic graphs. The dynamicity is represented as a multigraphs where two identical vertices can hold multiple edges for different times. In their work, the divided the multigraph into multiple window and perform degree distribution on different window graph. The work is formulated under the streaming paradigm where a batch of edges will arrive the system and gradually perform algorithm. Han and Sethu [21] have proposed an edge sampling algorithm for triangle counting of dynamic graphs.

Chen and Lui proposed a unified framework [9] to estimate the graphlet (small connected subgraph pattern) counts of the whole graph as well as the graphlet counts of individual nodes under the streaming graph model. To understand the structure of graph, Gabert *et al.* provided postmortem analysis to dense region in a dynamic graph using k-cores decomposition [18]. Previous streaming algorithms for k-core were designed [34].

Many centrality metrics can be used to find the important vertices in the graph, and multiple have been considered on dynamic graphs. Nathan and Bader [29] proposed a dynamic algorithm for updating Katz centrality in graphs under the streaming model. Under a streaming model, incrementally updating closeness centrality [36] and betweeness centrality [20] have also been studied.

3.3 Execution Model

There are three main ways to compute the many Pagerank values in the temporal model.

3.3.1 *Offline Pagerank Model.*

One can build independently a graph for each window and perform Pagerank. It requires reconstructing a correct graph from the *Event* data many times. The cost of the application will be driven by the cost of building the graphs, but the application becomes massively parallel since each time window can be computed independently. As such, this is an execution model that is appropriate for a massively distributed system such as a cloud platform.

3.3.2 *Streaming Pagerank Model.*

In the streaming model, the application maintain only a single copy of the graph. The version of the graph that is stored is meant to represent the graph as it is "now". Updates to the graph come as an edge stream. The streaming system needs to adjust the representation of the graph to account for the new edges and recompute the analysis accordingly. Middlewares have been built to support streaming graphs like STINGER [31] and ElGA [17]. These middlewares spend significant effort in maintaining a valid representation of the graph upon updates made to the graph by using advanced datastructures that minimize modification cost.

One of the benefit of streaming analysis is that when the calculation on the updated graph is made, the system has access to the result of the analysis on the previous version of the graph. This can lead to incremental algorithms which require less computation than recomputing the analysis from scratch [9, 20, 29, 34, 36].

We present now one way to incrementally update Pagerank values. A directed graph $G(V, E)$ with vertex and edge set V and E can be represented by a sparse matrix A where an edge($i \longrightarrow j$) is represented by $a_{ij} = 1$. If we represent the out degree of the graph by a diagonal matrix D then, *Pagerank* can be defined by the linear system [10, 19],

$$(I - \alpha A^T D^{-1})x = (1 - \alpha)v \tag{2}$$

Where α is the "teleportation" constant, v is the initial *Pagerank* vector usually filled by $1/|v|$ and x is the *Pagerank* vector. Jason presented [31] an approximation version of *Pagerank* for the streaming graph,

$$\Delta x^{k+1} = \alpha A_\Delta^T D_\Delta^{-1} \Delta X^k + \alpha(A_\Delta^T D_\Delta^{-1} - A^T D^{-1})x + r \tag{3}$$

where modifications of the streaming graph by edge addition or deletion are represented by Δ and k is represent the previous iteration. Here r is the residual error, $r = (1 - \alpha)v - (I - \alpha A^T D^{-1})x$.

The streaming execution model reduces the graph building time from offline execution. But they introduce more complex data structures to efficiently support insert and remove operations. The streaming model also enables to leverage incremental algorithms to decrease the total amount of computation. But it suffers from an inherent lack of parallelism. Since only one version of the graph is stored, the only available parallelism comes from the Pagerank computation itself and the graph updating procedure.

3.3.3 Postmortem Pagerank Model. We argue in this paper that in a postmortem model, we can produce analysis much faster than both the offline and streaming execution model.

Both offline and streaming models have significant graph construction cost, even though they are structured differently. In a postmortem model, we can build the graph representation in a single operation in a way that enable to access all the time windows.

The offline model benefits from high parallelism as it supports parallelism across different time-window and inside the kernel. The streaming model does not enable parallelism across time-window. But the postmortem model can support both levels of parallelism.

The streaming model leverages incremental updates to the Pagerank computation. Even if a postmortem execution leverages parallelism over different time-window, it can still arrange its calculation to leverage knowledge from the previous time-window if that information is known.

rowA = [0, 3, 9, 12, 16, 21, 24, 28]
colA = [2, 2, 3, 1, 1, 3, 4, 5, 7, 1, 2, 5, 5, 2, 6, 7, 2, 3, 3, 6, 7, 4, 5, 7, 2, 4, 5, 6]
timeA = [06/21/2021, 11/05/2021, 11/06/2021, 06/21/2021, 11/05/2021, 08/01/2021, 08/11/2021, 11/09/2021, 10/02/2021, 11/06/2021, 08/01/2021, 06/25/2021, 11/12/2021, 08/11/2021, 07/11/2021, 10/05/2021, 11/09/2021, 06/25/2021, 11/12/2021, 09/13/2021, 10/06/2021, 07/11/2021, 09/13/2021, 10/09/2021, 10/02/2021, 10/05/2021, 10/06/2021, 10/09/2021]

Figure 3: Temporal CSR Representation

4 POSTMORTEM GRAPH ANALYSIS

4.1 Data Representation

The performance of graph analyses vastly depends on the graph storage system. The offline and streaming model of computing Pagerank on a temporal graph suffer from data representation problem that can be addressed in a postmortem case. The CSR storage format is widely popular for the sparse matrices which is a fundamental attribute for Pagerank calculation using sparse-matrix vector multiplication (SpMV). We use a format that is similar derived from the CSR format.

Figure 3 shows a temporal CSR format for the graph presented in Figure 2b. Usually CSR requires two vectors, `rowA` and `colA`, to represent a graph. The `colA` vector is a concatenation of the adjacency list of the graph, while `rowA` indicates where the adjacency of each vertex starts. In other words, the first vertex neighbors are listed in `colA` between indices `rowA[0]` and `rowA[1]`. There are $V + 1$ entries in *rowA* and E entries in `colA`.

But for postmortem analysis we keep an additional vector which tracks timestamps for each edge, `timeA`, which will have the same size as the `colA` vector. There are duplicate entries in `colA`, because two vertices may appear multiple times in *Events* for different times. We store the neighbors of a vertex sorted by neighbors, and then by timestamp.

In this representation, we can iterate through the neighbors of vertices of a particular graph. For a particular vertex v of G_0 (for instance), the edges are all stored between `rowA[v]` and `rowA[v+1]`, but some of them do not exist for graph G_0. For a possible neighbor, the different times at which an event occured are stored consecutively in the temporal CSR representation. So as long as one of the edges has a timestamp between T_0 and $T_0 + \delta$, then it exists in G_0.

This basic representation requires one vector of size $V + 1$, and two vectors of size $|Events|$. One iteration of a Pagerank calculation requires performing one SpMV. This involves traversing the neighbors of every vertex and has a complexity of $\Theta(|Events|)$.

When the span of time increases or when δ decreases, the total number of events is not related to the total number of edges in one particular graph. Since $|Events|$ could be arbitrarily larger than the number of edges in any particular graph, the complexity of calculating a single SpMV can be arbitrarily larger than it should be.

To remedy this, we partition the representation in many multi-window graphs. Each multi-window graph represents a contiguous number of graphs and only stores the edges that are relevant to these graphs. We distribute the graphs uniformly to the multi-window graphs. So if the analysis involve X graphs and we represent the data with Y multi-window graph, each multi-window graph will contain $\frac{Y}{X}$ graphs.

A multi-window graph w has a vertex set V_w and an edge set E_w. Note that for a particular multi-window graph, V_w is typically smaller than the set of all vertices V since a vertex may not appear in that multi-window. Also, note that some edges may appear in two (or more) multi-window graph since an edge can appear in different consecutive graphs which could be in different multi-window graphs. In other words, this representation consumes more memory since $\sum_w |E_w| \geq |Events|$.

In this representation, performing SpMV for a graph only requires traversing the edges in the multi-window graph that contain the graph. And therefore computing SpMV for a graph in multi-window w has a complexity of $\Theta(|E_w|)$ which is closer to the number of edges in that graph than $\Theta(|Events|)$ is.

The question of how many multi-window graph remains to be investigated. We propose that a window graph should be accomodate by the system memory when computing Pagerank. The total memory cost of the representation is $encoding * (\sum_w |V_w| + 2 * |E_w|)$ where *encoding* accounts for the size of the number encoding (we use 64-bit for all data). And we need to retain memory available to store the intermediate data of Pagerank.

4.2 Partial Initialization

To calculate Pagerank, one needs to initialize the Pagerank values and the most common initialization value is $\frac{1}{|V|}$ where $|V|$ is the number of vertices in the graph. For us, the default would be $\frac{1}{|V|}$.

Now, the postmortem analysis is a sliding window process and two consecutive graphs share most of their vertices and in many case they share most of their edges. So the Pagerank values should be similar. And since Pagerank is a converging iterative process, having a better initial guess for the values should decrease the number of iterations to converge.

We build on out previous work [25] and propose a *partial initialization* for graph G_i that is a successor of window interval G_{i-1}. We denote by V_i all the vertices in graph G_i. We initialize the Pagerank of a vertex G_i simply based on the Pagerank of its neighbors that were present in G_{i-1} normalized to account for missing vertices. More specifically,:

$$PR_i[u] = \frac{|V_i \cap V_{i-1}|}{|V_i|} * \frac{PR_{i-1}[u]}{\sum_{v \in V_i \cap V_{i-1}} PR_{i-1}[v]} \tag{4}$$

Because the set of vertices encoded in a multi-window graph can be very different from the set encoded in the next multi-window graph, computing the indexing can be tedious. So we do not perform partial initialization across different multi-window graph. But since there are likely only few multi-window graph, the loss will be small.

We will experimentally validate the impact of partial initialization on convergence time.

4.3 Different Level Parallelization on Pagerank

We can utilize parallel computing at two different levels. We can parallelize over different time-window since they are all available in the postmortem representation, we call this *window-level parallelization*. We can also use parallelism inside the application kernel, here Pagerank, and we call this *application-level parallelization*. We can also leverage both at the same time which we call *nested parallelization*.

4.3.1 Window-Level Parallelization. Window-level parallelization is good for a well balanced graph and large number of time-window. If some graph are much larger than other ones, then window-level parallelization could lead to load imbalance. Also, if we have a limited number of time-window graph then we will only have a small amount of parallelism available.

Partial initialization may also be difficult to leverage in window-level parallelization. When starting to process graph G_i, one can only perform partial initialization if the Pagerank values of G_{i-1} are known by the thread. In practice, we implement the algorithm so if the same thread processes G_{i-1} and G_i, then partial initialization occurs.

Because of these two effects, a classic work scheduler is unlikely to be satisfactory. Think of OpenMP's classic dynamic scheduler. With a granularity of 1, it would likely allocate G_i and G_{i-1} always to different threads. This would result in the benefits of partial initialization being negated. A larger granularity would reduce the amount of parallelism available and takes the chance of having a single chunk of work contain graphs that are significantly larger than the rest of the chunks. And this would lead to load imbalance.

To remedy this problem, we opt for the workstealing scheduler of Intel TBB. With a workstealig scheduler, the threads will be originally allocated a chunk of contiguous work. That contiguous chunk will only be broken when the other threads are running out of work.

4.3.2 Application-Level Parallelization. In this model graphs are processed one at a time, in order from the first graph to the last graph. And all the parallelism happens inside the call to Pagerank for that particular graph. In this case, the parallelization is over the vertices of the graph.

In this model, we can use partial parallelization for every graph except first one of each multi-window graph.

This model will perform well if the workload in each graph significant compared to the total amount of work. In other words, we recommend using application-level parallelization for an instance with low number of graphs or where a few graphs carry most of the load of the analysis.

4.3.3 Nested Parallelization. Nested parallelism mixes both window-level and application-level parallelism. In other words, different graphs are performed in parallel and each Pagerank calculation is also performed in parallel. This mode of operation offers the most parallelism and is likely to provide benefits of both modes of operations.

This nested form of parallelism can be challenging for some parallel computing middleware. We use TBB and its workstealing scheduler to orchestrate the execution. This model will perform better in a large temporal graph with moderate number of graphs or well balanced window-application workload.

4.4 SpMM-inspired Postmortem Pagerank

When dealing with sparse matrices vector multiplication the primary bottleneck of executing the algorithm tend to come from moving the matrix from DRAM to the core, and from accessing the input vector in a random pattern. If the application supports it, it can be beneficial to execute multiple SpMVs simultaneously on different vectors and on the same matrix. One can perform multiple multiplication by reading the matrix only once. And interleaving the input vectors can transform the access patterns from mostly random to mostly regular. This is a common optimization in linear algebra: for instance LOBPCG tend to achieve higher performance than Lanczos to extract eigenvectors [39], and computing simultaneously multiple derivatives of radial basis functions [13].

Here, we have a similar structure. We compute Pagerank on multiple graphs but if the graphs are in the same multi-window graph, then the representation of the two graphs in memory is actually the same multi-window graph. By keeping in memory the intermediate value of multiple graphs' Pagerank calculation, we can perform one iteration of many Pageranks by accessing the multi-window graph only once. Also, since the graphs are likely sharing many common edges, the access pattern to the Pagerank vectors become also more regular.

We will abuse the name and refer to this method as an SpMM method. Even though technically, a different matrix is being used for the different Pageranks. We will also refer to the numbers of Pagerank being computed simultaneously as vector length by analogy to vector processing which plays a major role in SpMM implementation. Even though, the code may not actually use vectorization in practice.

Now, if we process consecutive graphs, say $G_0, G_1, \ldots, G_7$, then we are going to lose partial initialization. Indeed, the result of G_0 is needed to perform partial initialization on G_1. So we divided the multi-window graph into vector-length (e.g., 8) regions and picked first the graph from each region. This will perform first for instance $G_0, G_{10}, G_{20}, \ldots G_{70}$. These 8 Pageranks will not benefit from partial initialization. However, the next batch of graph processed will be $G_1, G_{11}, G_{21}, \ldots G_{71}$ which will all benefit from partial initialization.

We will investigate experimentally the impact of this SpMM-inspired optimization.

5 EXPERIMENTAL SETTINGS

5.1 Execution environment

All the experiments are performed on a node which is equipped with two Intel Xeon Gold model 6248R (Cascade Lake architecture, 24 cores per processor, no hyperthreading, 36MB L3 Cache) and 384GB GB of DDR4 memory. The operating system used in the machine is Linux `3.10.0`.

All the codes are writen in C++ and compiled by the Intel C++ compiler `icpc` version `19.1.3.304`. Codes are compiled with optimization flag `-O3` and `xCORE-AVX512` flags, so the compiler generates a binary optimized for the architecture.

All the streaming version of *Pagerank* are performed on the `STINGER` [31] framework. `STINGER` is a package designed to support streaming graph analytics by using in-memory parallel computation to accelerate the computation. `STINGER` supports Pagerank with an incremental algorithm. The only modifications to `STINGER` that we performed are to the edge event injection logic so as to updates in batches equivalent to the postmortem code. This makes the code bases produce the same results and makes the comparison fair.

Table 1: Graphs and Parameters

Name(Events)	Sliding Offset	Window Size
ca-cit-HepTh (2,673,133)	12 hours, 1, 2 days	10, 15, 90, 180, 730, 1460 days
stackoverflow (47,903,266)	12 hours, 1 day	10, 15, 90, 180, 730 days
askubuntu (726,661)		90, 180 days
Youtube-Growth (12,223,774)		60, 90 days
epinions-user-ratings (13,668,281)		
ia-enron-email (1,134,990)	12 hours, 2 days	2, 4 days
wiki-talk (6,100,538)	12 hours, 1, 2, 4 days	10, 15, 90, 180 days

5.2 Graphs

We perform our experiments on real-world data sets to avoid the bias introduced by random graph generator. We select graphs from the *Stanford Large Network Dataset Collection* (SNAP) [27], network repository, and *DIMACS* [2, 33] data sets that are frequently used in graph algorithm research.

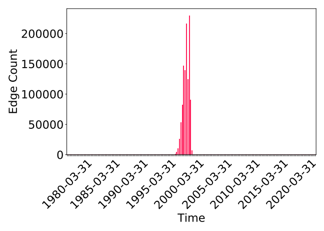
Table 1 presents the temporal graphs and provide details of the application parameters (window size, and window offset) that we set. We picked parameters that would look at the data at different scale and resolution. Still we choose to have the time-windows overlap (all the graph share some edges from its previous graph) since it seems likely analysis would always want that property. We assume all the edges of the graphs are sorted in non-decreasing order of their arrival time.

6 RESULTS

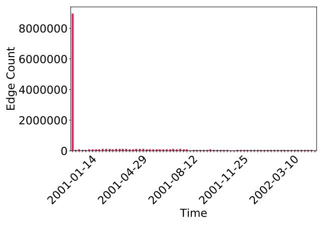
6.1 Edge Distribution of Temporal Graph

Figure 4 presents the edge distribution for all the graphs over time. We can see the patterns of temporal edges are different for different graphs. This provides a diversity of instances to test our methods.

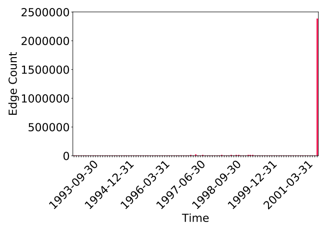
Figure 4a shows the the email communication of the *Enron Corpus* where we can see some big spike around 2001. These spikes represents the period of time when Enron scandal happened which is the period of time mostly captured by the dataset. Figure 4b shows the user review ratings collected by Epinions. Epinions was established in 1999 and peaked around 2001 and later they acquired by eBay. It is a bipartite graph, an edge represents a user reviewing a product. We can see around 2001 user reviews shows a huge spikes which is the reason the company was acquired. Citation graph `ca-cit-HepTh4c` also shows an irregular distribution pattern of temporal edges. On these networks, the Pagerank calculation bottleneck will be on a few graphs since few time-window cover most the edges in the dataset. We will see that this distribution of work will make application-level parallelism more efficient than window-level parallelization.



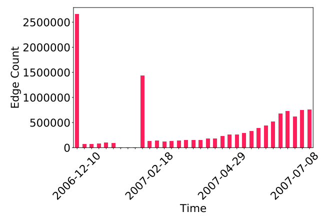
(a) ia-enron-email



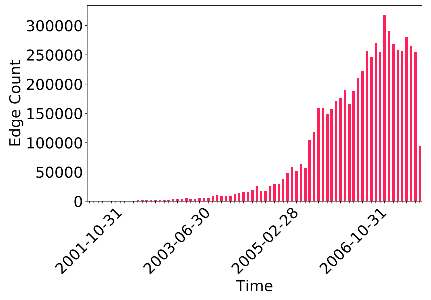
(b) epinions-user



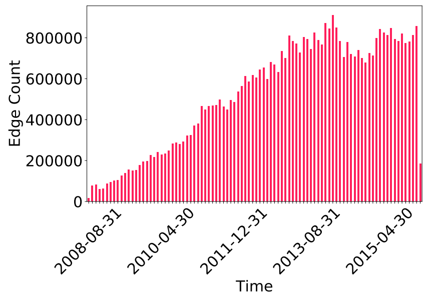
(c) ca-cit-HepTh



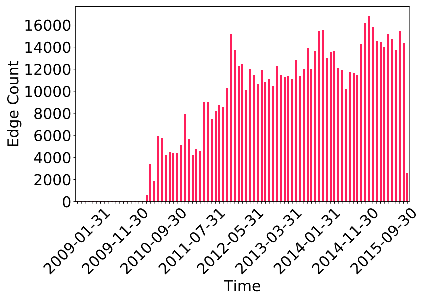
(d) youtube-growth



(e) wigki-talk



(f) stackoverflow



(g) askubuntu

Figure 4: Temporal graph edge distribution over the time period.

The temporal edge distribution for `wiki-talk` (Figure 4e), `askubuntu` (Figure 4g), and `stackoverflow` (Figure 4f) show increasing amount of streaming edges over time. But the number of edges that come in is relatively smooth. Graphs with balanced high-volume edges with large number of windows are well suited for nested parallelism.

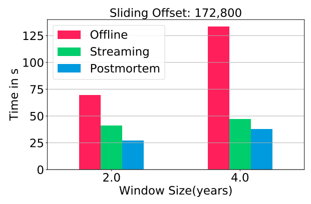
`youtube-growth` 4d shows a pattern that is both bursty by moment but steady in general.

6.2 Postmortem is usually faster than Offline and Streaming

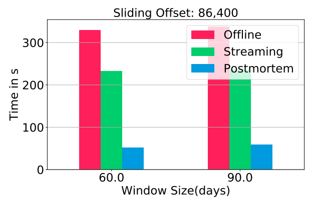
We compare the performance of the three execution models: Offline, Streaming and Postmortem. Postmortem here uses partial initialization and each temporal graph is partitioning into 6 multi-window graphs. Postmortem uses only an application-level parallelism with a static scheduler. In other words, this is a bare-bone postmortem computation where the execution parameters have not been tuned.

Figure 5 shows the comparison among Naive, Streaming and Postmortem Pagerank for some of the temporal graphs from four of our temporal datasets. The performance on `enron-email` is reported in Figure 5a. The Streaming version is faster than the offline version. But Postmortem outperforms both of them. Figure 5b shows the performance for the youtube dataset. On this graph as well, streaming is faster than offline; and postmortem is faster than both. The postmortem version outperforms Streaming by more than 3 times on that dataset.

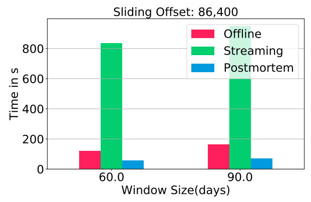
Figure 5c shows the performance for the epinions dataset. On that dataset, streaming is much slower than both offline and postmortem. Postmortem is faster than both and about more than 40 times faster than streaming. Figure 5d shows results on the `wikitalk` dataset. Here streaming is also slower than both other methods. Postmortem is slightly slower than offline on small window size and better on larger ones.



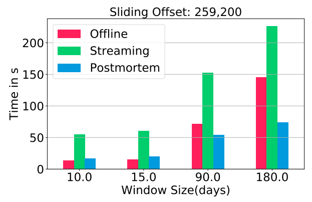
(a) Enron



(b) YouTube



(c) Epinion



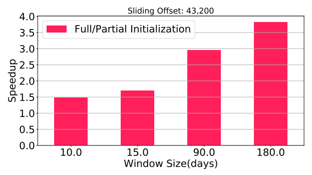
(d) Wikitalk

Figure 5: Performance of Naive, Streaming and Postmortem Pagerank

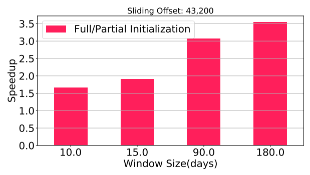
6.3 Postmortem Detailed Results

6.3.1 Impact of Partial initialization. Figure 6 presents the impact of partial initialization on `stackoverflow` and `wiki-talk` temporal graph. It shows a performance gain that correlates with the size of the window and ranging from being 1.5 times faster to 3.5 times faster. It makes intuitive sense that the smart initialization improves the performance more on larger windows since the successive graphs become more similar.

We found similar speedup for other experimental graphs also (not shown). And from now, we will show results with partial initalization only rather than full initialization.



(a) Stackoverflow



(b) Wiki-talk

Figure 6: Impact of partial initialization on postmortem graph analysis.

6.3.2 *Partitioner and Granularity.*

We saw that there is imbalance in the distribution of edges over time. But we also know that social graphs have power law edge distribution which makes the degree of the graph very unbalanced. As a result the bottleneck of the application is often the time-window graph that has many more edges than the other ones, and the block of vertices in the graph with extremely high degree.

In our experiment, we choose Intel's Thread Building Block(TBB) mechanism to parallelize the postmortem Pagerank to benefit from its scheduler. Now, TBB provide multiple `partitioners` and support different granularity. The `auto_partitioner` is the default workstealing scheduler while `simple_partitioner` is a variant of it. TBB also provides a `static_partitioner` which does not benefit from workstealing.

Choosing granularity requires experimental analysis. It depends on the partitioners, system cache memory, problem size, *etc.* We perform our experiments using a variety of granularity sizes to figure out the behavior of the results for certain attributes.

Figure 7 presents the performance of Pagerank on `wiki-talk` for different partitioner and granularity size for a certain sliding window and window size. The window size of the graph is 256 that means we can split the window-level parallelization at maximum by 256 where each worker thread will receive a single window. And we can see a performance drop after 128 for window-level parallelization because it lacks of parallelism. Nested and Pagerank-level parallelization show better result than window-level but they also lost some performance gain. The main reason also high granularity size assign large number of windows to each worker thread and make it imbalanced.

Overall, the performance of the `static_partitioner` seems worse than that of the other two partitioners. And the auto and simple partitioner are fairly comparable in performance.

6.3.3 *Impact of the number of Multi-Window Graphs.*

The number of multi-window is an important parameter. If the number is too low, there runtime overhead due to traversing edges out of the graph the algorithm is considering will be high. If the number is too high, the system wastes memory and the impact of partial initialization will be lower.

The results presented in Figure 8 show that one the number of multi-window is "large enough", the performance no longer varies.

6.3.4 *Comparing SpMV to SpMM.*

The main difference between the *SpMV* and *SpMM* versions of postmortem Pagerank is that the SpMM version computes multiple Pagerank vector at once in a multi-window graph and treat them as a matrix. We choose a number of vector of either 8 or 16. Choosing a high number of vector in SpMM will reduce benefit of the partial initalization because all the initial Pagerank vectors will do full initialization.

Figure 7 shows postmortem Pagerank performance on `wiki-talk`, where we can see the number of windows is 256. Our experimental results show that SpMM is usually much faster than SpMV.

6.3.5 *Which level of parallelization?*

Figure 9 shows better performance for Pagerank-level and nested but shows lack performance of the window-level parallelism. The main reason is the number of windows is only 6 where we have 48 available processors which stiffles the performance of window-level parallelism.

Figure 10 shows godd performance for window-level paralleization because of large number of windows. On the other hand at Figure 7 show better performance for nested parallelization.

Application-level parallelization is well suited for the well balanced windows with large window size graph. On the other hand window-level parallelization can out-perform other on the occasion where number of windows is large but the number of window size is smaller. That means less work in application level. Nested always show optimal or near optimal performance because it can adapt to both form of available parallelism.

6.3.6 *Best Mechanism and suggest parameters.*

Figure 11 shows the overall best performance by the postmortem Pagerank relatively to the streaming model over the different configurations we tested. The Postmortem model proved to be between 50 and 800 times fater than the streaming model.

However, a user may not know how to set parameters. We provide a simple rules to set them that should lead to decent performance. Our experiments show that SpMM is never a bad choice. For partitioner, `auto_partitioner` with granularity size under 4 usually provides good results. To chose the type of parallelism, one need to look at the load balance in edges of different time windows. Unless the workload is dominated by couple of windows or very small number of multi-window, nested parallelization is the good fit for almost every graph.

We generated the performance of following this guidelines on `wiki-talk` across different sliding offset and window size and reported the results in Figure 12. The configuration does not report the best performance but reports very honorable performance at little tuning cost.

7 CONCLUSION

The study of performance of temporal graph analysis is often considered mostly in the streaming model where one wants to maintain the analysis current with the most recent data. However an other common use case is to analyse a temporal data postmortem once all the data is known. We showed in this paper how to perform Pagerank efficiently on modern parallel systems by leveraging data representation, incremental algorithms, and different types of parallelism. When using these techniques, a postmortem analysis can be conducted from 50 to 800 times faster than a streaming analysis.

The methods we presented can still be refined: multiple questions remain. We partitioned the temporal data in multi-windows with equal number of graphs, but this may not be the decomposition that minimize memory and work overheads. We only considered

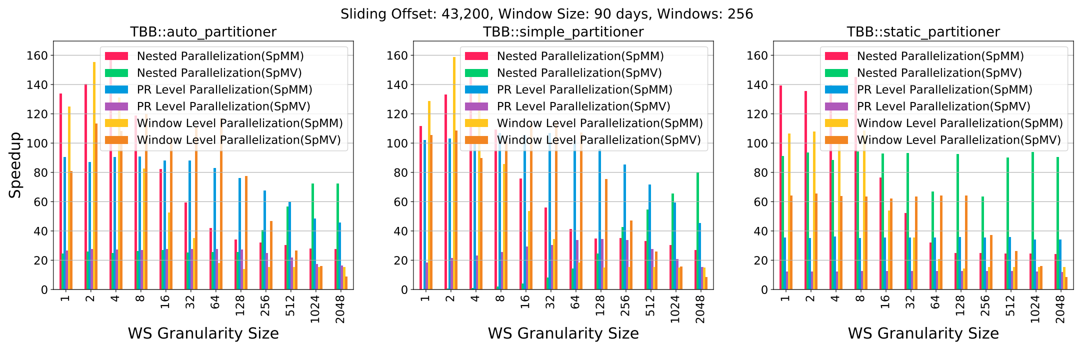


Figure 7: Postmortem Pagerank comparison over streaming on `wiki-talk` graph (SpMM load 16 Pagerank vectors).

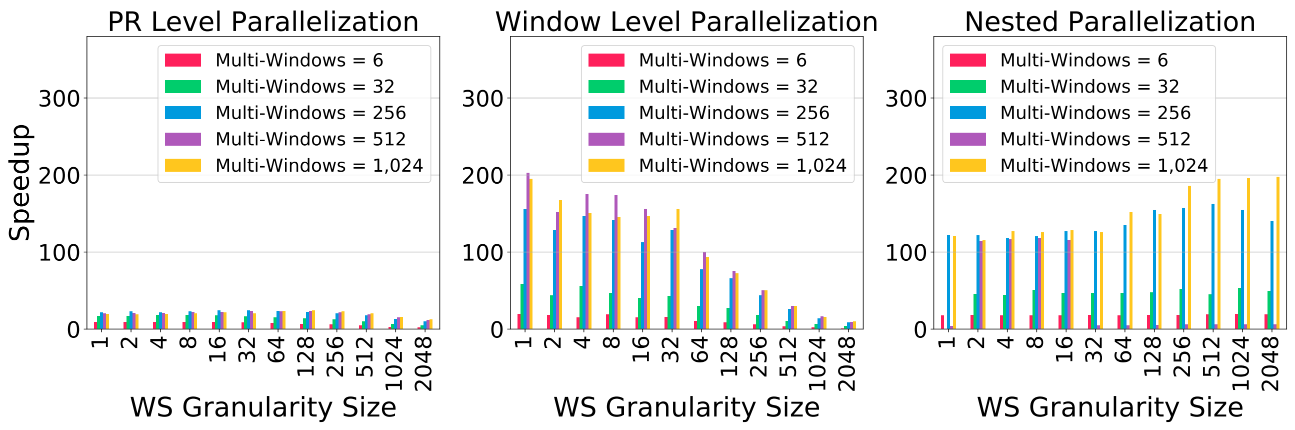


Figure 8: Postmortem Pagerank performance using TBB `auto_partitioner` for wiki-talk network for different number of multi-window.

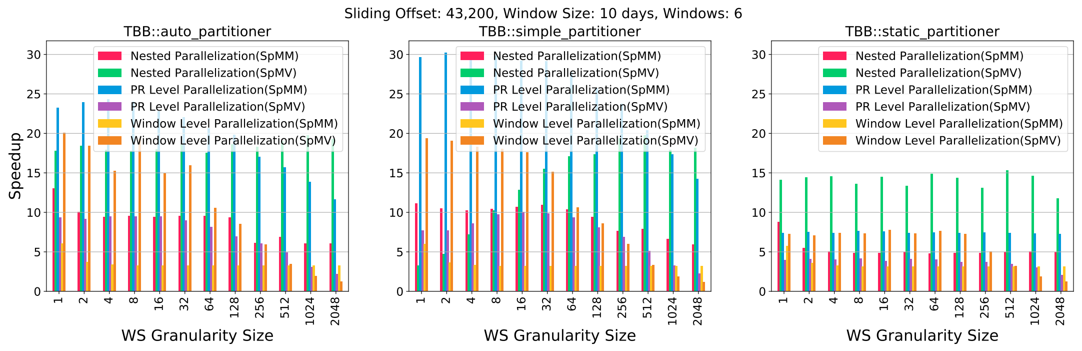


Figure 9: Postmortem Pagerank comparison over streaming on `wiki-talk` graph (SpMM load 16 Pagerank vectors).

Pagerank, but other analysis, like centralities for instance, behave in less regular way when small changes impact the graph. Nowadays, much graph analysis is performed on GPU-enabled system or on distributed memory systems; and extending our techniques to such systems would make temporal analysis more practical.

ACKNOWLEDGMENTS

This work is supported by grant from the National Science Foundation CCF-1652442 and was made possible by a computing allocation given by TACC through XSEDE.

REFERENCES

[1] Zainab Abbas, Paolo Sottovia, Mohamad Al Hajj Hassan, Daniele Foroni, and Stefano Bortoli. 2020. Real-time Traffic Jam Detection and Congestion Reduction Using Streaming Graph Analytics. In *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 3109–3118.

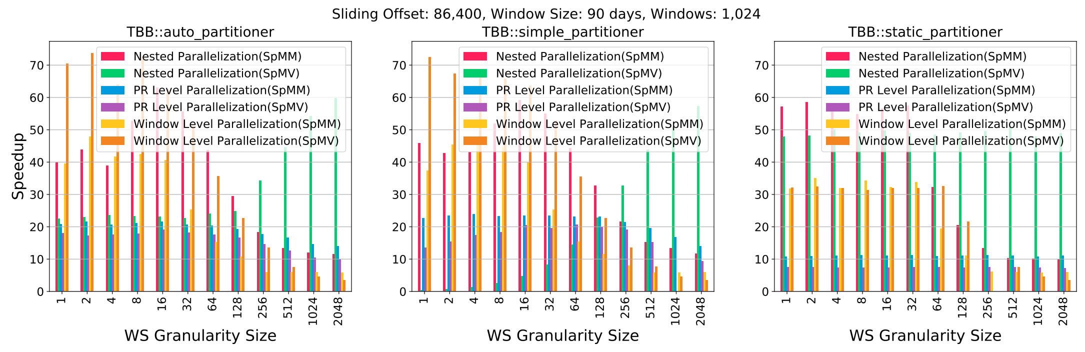
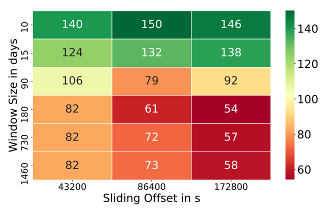
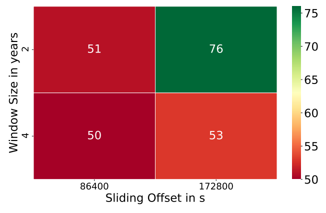


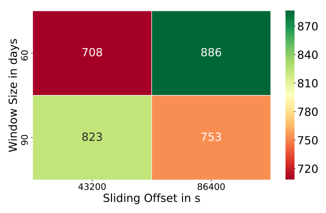
Figure 10: Postmortem Pagerank comparison over streaming on `wiki-talk` graph (SpMM load 16 Pagerank vectors).



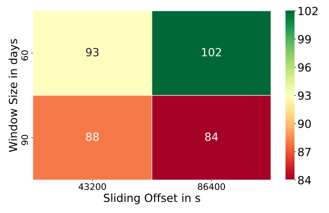
(a) ca-cit-HepTh



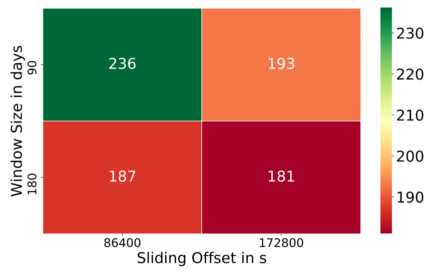
(b) enron corpus



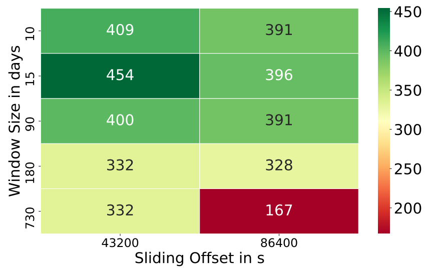
(c) Epinions User Ratings



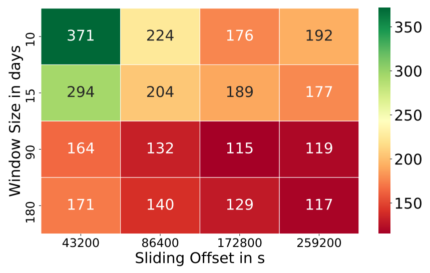
(d) Youtube-Growth



(e) Askubuntu



(f) Stackoverflow



(g) wiki-talk

Figure 11: Best performance gain by postmortem Pagerank over streaming version.

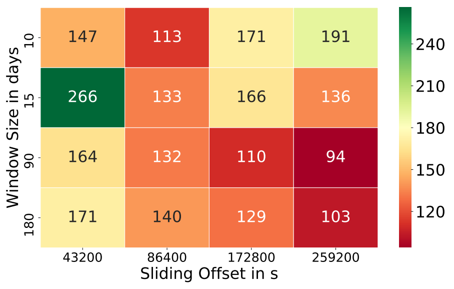


Figure 12: Postmortem performance with suggested parameter on wikitalk.

[2] David A Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. 2013. *Graph partitioning and graph clustering*. Vol. 588. American Mathematical Society Providence, RI.
[3] Alex Bavelas. 1950. Communication patterns in task-oriented groups. *The journal of the acoustical society of America* 22, 6 (1950), 725–730.
[4] Scott Beamer, Krste Asanović, and David Patterson. 2017. Reducing pagerank communication via propagation blocking. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 820–831.
[5] Ala Berzinji, Lisa Kaati, and Ahmed Rezine. 2012. Detecting key players in terrorist networks. In *2012 European Intelligence and Security Informatics Conference*. IEEE, 297–302.
[6] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment* 2008, 10 (2008), P10008.
[7] Tami Carpenter, George Karakostas, and David Shallcross. 2002. Practical issues and algorithms for analyzing terrorist networks. In *Proceedings of the western simulation multiconference*.
[8] Rohan Chandra, Tianrui Guan, Srujan Panuganti, Trisha Mittal, Uttaran Bhattacharya, Aniket Bera, and Dinesh Manocha. 2020. Forecasting trajectory and behavior of road-agents using spectral clustering in graph-lstms. *IEEE Robotics and Automation Letters* 5, 3 (2020), 4882–4890.
[9] Xiaowei Chen and John CS Lui. 2017. A unified framework to estimate global and local graphlet counts for streaming graphs. In *Proc. ASONAM*. 131–138.
[10] Gianna M Del Corso, Antonio Gulli, and Francesco Romani. 2005. Fast PageRank computation via a sparse linear system. *Internet Mathematics* 2, 3 (2005), 251–273.
[11] Prasanna Desikan, Nishith Pathak, Jaideep Srivastava, and Vipin Kumar. 2005. Incremental Page Rank Computation on Evolving Graphs. In *Special Interest Tracks and Posters of WWW*. 1094–1095.
[12] R Devika and V Subramaniyaswamy. 2021. A semantic graph-based keyword extraction model using ranking method on big social data. *Wireless Networks* 27, 8 (2021), 5447–5459.

[13] Gordon Erlebacher, Erik Saule, Natasha Flyer, and Evan Bollig. 2014. Acceleration of Derivative Calculations with Application to Radial Basis Function - Finite-Differences on the Intel MIC Architecture. In *Proc. of International Conference on Supercomputing (ICS)*.

[14] Hossein Esfandiari, Silvio Lattanzi, and Vahab Mirrokni. 2018. Parallel and streaming algorithms for k-core decomposition. In *International Conference on Machine Learning*. PMLR, 1397–1406.

[15] Leila Eskandari, Jason Mair, Zhiyi Huang, and David Eyers. 2018. T3-Scheduler: A topology and traffic aware two-level scheduler for stream processing systems in a heterogeneous cluster. *Future Generation Computer Systems* 89 (2018), 617–632.

[16] Linton C Freeman. 1977. A set of measures of centrality based on betweenness. *Sociometry* (1977), 35–41.

[17] Kasimir Gabert, Kaan Sancak, M. Yusuf Özkaya, Ali Pinar, and Ümit V. Çatalyürek. 2021. ElGA: Elastic and Scalable Dynamic Graph Analysis. In *Proc. SC (SC '21)*. Article 50, 15 pages.

[18] Kasimir Georg Gabert, Ali Pinar, and Umit Catalyurek. 2020. *Finding Dense Areas of Massive Changing Graphs.* Technical Report. Sandia National Lab.(SNL-NM).

[19] David Gleich, Leonid Zhukov, and Pavel Berkhin. 2004. Fast parallel PageRank: A linear system approach. *Yahoo! Research Technical Report YRL-2004-038, available via http://research. yahoo. com/publication/YRL-2004-038. pdf* 13 (2004), 22.

[20] Oded Green, Robert McColl, and David A. Bader. 2012. A Fast Algorithm for Streaming Betweenness Centrality. In *International Conference on Privacy, Security, Risk and Trust and International Conference on Social Computing*. 11–20.

[21] Guyue Han and Harish Sethu. 2017. Edge sample and discard: A new algorithm for counting triangles in large dynamic graphs. In *Proc. ASONAM*. IEEE, 44–49.

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

[23] Liaquat Hossain, Shahriar Tanvir Murshed, and Shahadat Uddin. 2013. Communication network dynamics during organizational crisis. *Journal of Informetrics* 7, 1 (2013), 16–35.

[24] Md. Maruf Hossain and Erik Saule. 2021. Impact of AVX-512 Instructions on Graph Partitioning Problems. In *ICPP Workshops 2021: 50th International Conference on Parallel Processing, Virtual Event / Lemont (near Chicago), IL, USA, August 9-12, 2021*, Federico Silla and Osni Marques (Eds.). ACM, 33:1–33:9. https://doi.org/10.1145/3458744.3473362

[25] Md Maruf Hossain and Erik Saule. 2021. Postmortem Graph Analysis on the Temporal Graph. In *ICPP poster 2021: 50th International Conference on Parallel Processing*.

[26] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2005. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proc. of SIGKDD*. 177–187.

[27] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[28] Muhammad Ali Masood and Rabeeh Ayaz Abbasi. 2021. Using graph embedding and machine learning to identify rebels on twitter. *Journal of Informetrics* 15, 1 (2021), 101121.

[29] Eisha Nathan and David A Bader. 2017. A dynamic algorithm for updating katz centrality in graphs. In *Proc. ASONAM*. 149–154.

[30] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[31] Jason Riedy. 2016. Updating pagerank for streaming graphs. In *Proc. IPDPSW*. IEEE, 877–884.

[32] Zafar Saeed, Rabeeh Ayaz Abbasi, Abida Sadaf, Muhammad Imran Razzak, and Guandong Xu. 2018. Text stream to temporal network-a dynamic heartbeat graph to detect emerging events on twitter. In *Proc. PAKDD*. 534–545.

[33] Peter Sanders, Christian Schulz, and Dorothea Wagner. 2014. Benchmarking for graph clustering and partitioning. In *Encyclopedia of Social Network Analysis and Mining*, R. Alhajj and J. Rokne (Eds.). Springer.

[34] Ahmet Erdem Sariyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V Çatalyürek. 2013. Streaming algorithms for k-core decomposition. *Proceedings of the VLDB Endowment* 6, 6 (2013), 433–444.

[35] Ahmet Erdem Sariyüce, Kamer Kaya, Erik Saule, and Ümit V Çatalyiirek. 2013. Incremental algorithms for closeness centrality. In *IEEE Big Data*. 487–492.

[36] Ahmet Erdem Sarıyüce, Erik Saule, Kamer Kaya, and Ümit V Çatalyürek. 2015. Incremental closeness centrality in distributed memory. *Parallel Comput.* 47 (2015), 3–18.

[37] Andrew Stolman and Kevin Matulef. 2017. HyperHeadTail: a streaming algorithm for estimating the degree distribution of dynamic multigraphs. In *Proc. ASONAM*. 31–39.

[38] Lei Yang, Lei Qi, Yan-Ping Zhao, Bin Gao, and Tie-Yan Liu. 2007. Link analysis using time series of web graphs. In *Proc. of CIKM*. 1011–1014.

[39] Zheng Zhou, Erik Saule, Hasan Metin Aktulga, Chao Yang, Esmond G. Ng, Pieter Maris, James P. Vary, and Ümit V. Çatalyürek. 2012. An Out-of-core Eigensolver on SSD-equipped Clusters. In *Proc. of IEEE Cluster*.