Chronos: A Graph Engine for Temporal Graph Analysis

Wentao Han[†§] Youshan Miao[‡§] Kaiwei Li[†§] Ming Wu[§] Fan Yang[§] Lidong Zhou[§]
Vijayan Prabhakaran[§] Wenguang Chen[†] Enhong Chen[‡]

[†]Tsinghua University [‡] University of Science and Technology of China [§] Microsoft Research

Abstract

Temporal graphs capture changes in graphs over time and are becoming a subject that attracts increasing interest from the research communities, for example, to understand temporal characteristics of social interactions on a time-evolving social graph. Chronos is a storage and execution engine designed and optimized specifically for running in-memory iterative graph computation on temporal graphs. Locality is at the center of the Chronos design, where the in-memory layout of temporal graphs and the scheduling of the iterative computation on temporal graphs are carefully designed, so that common "bulk" operations on temporal graphs are scheduled to maximize the benefit of in-memory data locality. The design of Chronos further explores the interesting interplay among locality, parallelism, and incremental computation in supporting common mining tasks on temporal graphs. The result is a high-performance temporal-graph system that offers up to an order of magnitude speedup for temporal iterative graph mining compared to a straightforward application of existing graph engines on a series of snapshots.

1. Introduction

Graphs can naturally capture connections and relationships. Real world graphs often evolve over time as the connections and relationships change [13]. There is growing interest in analyzing not only the static structure of graphs, but also their time-evolving properties. For example, to study diameter changes of an evolving social network [13], to characterize social relationships according to changing user activities in online social networks [32], and to observe how webpage ranks change over time [33]. New algorithms are also proposed specifically for evolving graphs to extract new insights [12, 32].

Many graph mining algorithms are designed for a static graph, e.g., to compute PageRank or weakly connected components. Understanding the evolution of graphs over time often involves running those graph mining algorithms on a series of *snapshots*, which we refer to as *temporal graph mining*. Here a snapshot corresponds to the static graph at a particular time point. Chronos is a parallel in-memory graph engine designed to enable efficient temporal graph mining both on multi-core machines and in distributed settings. The addition of the time dimension in Chronos gives rise to interesting and rich new opportunities, beyond the existing graph engines that work on static graphs [7, 16, 17, 23, 27].

Fundamentally, the design of Chronos centers on two issues: the in-memory layout of a temporal graph and the scheduling of the graph computation. Rather than taking the straightforward approach of applying an existing graph engine on each snapshot of a temporal graph, Chronos proposes *locality-aware batch scheduling* (LABS) with the following two key observations.

First, locality in data layout matters greatly in graph computation. For a temporal graph, the data layout can exhibit either *time(-dimension) locality*, where the states of a vertex (or an edge) at two consecutive time points are laid out consecutively, or *structure(-dimension) locality*, where the states of two neighboring vertices at the same time point (i.e., in the same graph snapshot) are laid out close to each other. Time-locality and structure-locality are not created equal: time-locality can be arranged "perfectly" because time progresses linearly, but structure-locality can only be approximate because it is challenging to project a graph structure into a linear space. The design of Chronos therefore favors time-locality when it lays out multiple snapshots of a graph in memory.

Second, Chronos schedules graph computation to leverage the time-locality in the in-memory temporal graph layout. Such co-design of scheduling and layout is essential to maximize the benefits of locality. Traditional graph engines arrange computation around each vertex (or each edge) in a graph; while temporal graph engines, in addition, calculate the result across multiple snapshots. Chronos makes a deci-

EuroSys 2014, April 13–16 2014, Amsterdam, Netherlands.
Copyright © 2014 ACM 978-1-4503-2704-6/14/04. . . $15.00.
http://dx.doi.org/10.1145/2592798.2592799

sion to *batch* operations associated with each vertex (or each edge) across multiple snapshots, instead of batching operations for vertices/edges within a certain snapshot.

These seemingly simple observations have proven effective on different graph computing implementations, whether it is stream-based, push-based, or pull-based, as described with more details in Section 5.

To execute efficiently on a multi-core server, the design of Chronos further considers issues related to parallelism, such as the impact of locking to avoid conflicts. Chronos also examines the interplay with incremental graph computation that might help speed up iterative graph mining on multiple snapshots [5].

We perform extensive evaluations using 5 graph computation algorithms on 4 real-world temporal graphs with billions of edges and hundreds of millions of vertices. The result shows the effectiveness of Chronos: with LABS, Chronos can achieve more than 10 times the speedup on a single-thread execution, compared to the straightforward approach of running the same graph mining algorithm on each snapshot. On a 16-core machine, Chronos outperforms the embarrassingly parallel execution of the same graph mining on multiple snapshots by more than a factor of 2 due to LABS despite the need for locking in Chronos. The benefits of LABS extend to our small distributed deployment of Chronos. Our in-depth analysis reports details such as cache/TLB miss counts, overheads from lock contentions, inter-core communication overheads, to reveal the underlying reasons for the superior performance of Chronos.

In summary, the paper makes the following contributions: 1) a novel scheme to jointly design the data layout and scheduling for temporal graph; 2) a complete system to implement the ideas and demonstrate the significant performance improvement; 3) an in-depth evaluation and comparison of alternative design choices.

2. Temporal Graph Mining Overview

A temporal graph tracks *all* the information relevant to the evolution of a graph, including every graph edit activity, such as addition of a vertex or deletion of an edge, along with its timestamp. Such information may grow infinitely over time, hence Chronos assumes that the temporal graph data is initially available in persistent storage such as disk or flash. Before the temporal graph computation, Chronos extracts the on-disk data into the desired in-memory layout, we defer the discussion of the on-disk layout to Section 4.

2.1 Motivating Examples

Chronos supports all applications that are also targets of existing graph engines [7, 16, 17, 23, 27] with the additional graph mining capability in the time dimension. This includes the graph mining at some point-in-time or within a time range.

One example of point-in-time graph mining is to compute the diameter of a graph at time t, which involves traversing the graph snapshot at t to find the longest shortest path. Current graph engine can handle point-in-time analysis only at “now”.

An example of graph mining in a time range is to study the change of the PageRank of each vertex over a given period of time. This often involves computation on a series of graph snapshots at a few given time points within the given time range. Such graph mining in a time range receives more attention from the data mining community [13] and hence is the focus of Chronos.

2.2 Temporal Iterative Graph Computation

Existing graph engines support static graph mining with a *scatter-gather* iterative computation model [7, 16, 17, 23, 27]. Such a model performs computation in multiple iterations. Each iteration includes a *scatter* phase that propagates a local value (e.g., a rank) associated with a vertex to its neighbors, followed by a *gather* phase that accumulates updates from neighbors to compute the new local value of a vertex.

In order to understand the evolution of graphs over time (e.g., to understand how rank values of web pages were changing over a period of time), users need to run scatter-gather iterative computation on a series of graph snapshots representing the states of a temporal graph at different points in time. Supporting such *temporal iterative graph mining* efficiently is the design goal of Chronos.

At first glance, it might seem that the straightforward way of applying an existing scatter-gather graph engine on each snapshot, which we use as the baseline for our evaluations, would simply work. Our observation suggests that this straightforward solution is far from optimal and leaves significant room for performance improvement.

To support iterative graph computation over a series of snapshots, Chronos must address the following design issues.

First, it must decide on the in-memory layout of snapshot series. Our experiences with real world large temporal graphs have shown that different in-memory data structure leads to great difference in the performance of graph computation (see Section 6.1). A series of graph snapshots have two dimensions: one is the time dimension across snapshots and the other is the graph-structure dimension among neighbors within a snapshot.

Second, Chronos must decide how to schedule the computation. The computation spans multiple snapshots and, for each snapshot, involves propagation among neighbors within that snapshot. There are different ways of scheduling the computation. One obvious choice is to schedule iterative computation for each snapshot, while an alternative is to run iterative computation on multiple snapshots together. That is, the propagation from one vertex to its neighbors on multiple snapshots can be scheduled together.

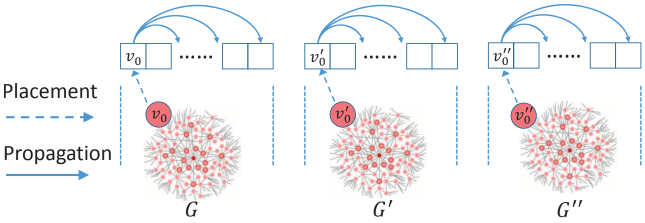


Figure 1. Propagation pattern when data is grouped by snapshot.

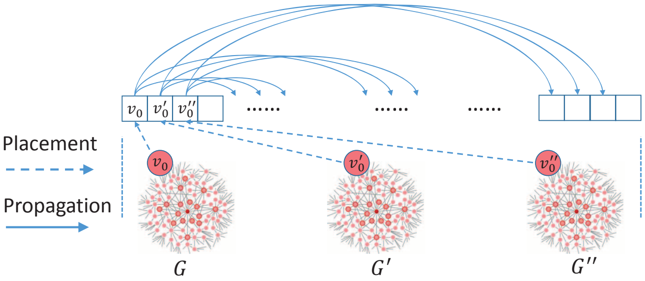


Figure 2. Propagation pattern when data is grouped by vertex.

Third, Chronos must enable parallelism of the computation on multi-core or distributed machines. One obvious option is to assign different snapshots to different cores, while an alternative is to assign different graph partitions (aligned across multiple snapshots) to different cores. While the former exhibits embarrassing parallelism since no synchronization required in the computation, our results show that the latter strategy, when carefully designed, can produce noticeably better performance.

Finally, Chronos should also consider the effect of incremental computation, where it is sometimes feasible and effective to continue computing from the result of the previous snapshot, especially when the computation is proportional to the numbers of changes between two snapshots and significantly lower than computing from scratch.

These design choices have intrinsic dependencies among them and have to be considered together.

3. Chronos Design

Chronos proposes *Locality-Aware Batch Scheduling* (LABS), which makes two fundamental design choices: one is to favor time locality over structure locality when laying out a temporal graph; the other is to match scheduled access pattern with data locality.

3.1 LABS Illustrated

Figures 1 and 2 illustrate the opportunity brought by LABS on achieving better data locality. The figures show a temporal graph with three graph snapshots, G, G', G'', representing the graph states at three different points of time. v_0, v'_0, and v''_0 are the three versions of the same vertex in the 3 snapshots, respectively (for simplicity, we use the same notations to denote the associated data of the vertex). The vertex has three neighbors and thus needs to propagate its value to the three during the graph computation.

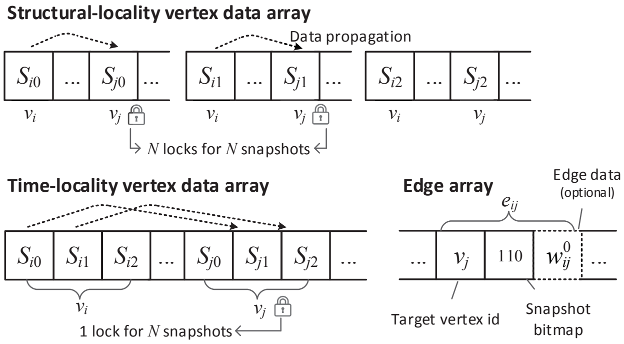


Figure 3. Illustration of time-locality and structure-locality in-memory layouts for vertices and edges.

Figure 1 shows a straightforward data organization for a temporal graph. It arranges the data snapshot by snapshot. This arrangement, at the expense of scattering different versions of data of the same vertex (say, v_0) away, expects to place all vertices within the same snapshot close to each other. However, in a graph structure, it is difficult to ensure all the neighbors are placed close to every vertex. If unfortunately all the three neighbors of v_0 are scattered far from each other, running graph mining snapshot by snapshot would incur 9 cache misses for the propagations from v_0 to the 3 neighbors in the 3 snapshots within each iteration.

On the other hand, Figure 2 shows another data layout of the temporal graph, which groups graph data by vertex. LABS adopts this layout to place different versions of data of the same vertex together. Through batching the propagations to multiple versions of the same neighbor of v_0, LABS can decrease the number of cache misses to 3 if the vertex data size is small enough to fit in the cache line.

In addition to the reduced cache misses, LABS also has the advantage of decreasing the data access volumes and offers interesting design choices for parallel graph computation. The following subsections describe the details of Chronos' LABS design.

3.2 In-Memory Data Structure

At the beginning of the computation, Chronos loads the on-disk data that contains the graph snapshots of interest into the main memory for repeated accesses. The information is divided into an edge array and a vertex data array (for application data, such as ranks, associated with each vertex).

The in-memory data structure maintains the reconstructed states at the specified snapshots and discards any unnecessary fields (e.g., timestamps) stored in the on-disk layout. Our experience show that the in-memory computation often dominates the end-to-end cost of graph mining. We therefore focus on optimizing the in-memory data structure.

To lay out a temporal graph in memory, we have the choice of favoring locality in the *graph structure*, or locality in the *time dimension*. We name the former *structure-locality* layout and the latter *time-locality* layout.

As shown at the top of Figure 3, the structure-locality layout groups the graph information in the same snapshot together in a way that maximizes structure locality [23]. It places the data one snapshot after another, which favors locality on the graph structure. Suppose the computation propagates S_{ik}, the state of vertex v_i in snapshot k, to S_{jk}, vertex v_j in the same snapshot (designated by the dashed arrow), this structure-locality layout tries to place S_{ik} close to S_{jk}.

The time-locality layout, instead, groups the information for the same vertex across multiple snapshots and places the data one vertex after another. As shown in Figure 3, the vertex data in the same snapshot is scattered in this layout, compared to the structure-locality layout. Yet, the time-locality layout exhibits good data locality in the time dimension. In Figure 3, S_{i1}, the data of vertex v_i in snapshot 1, will *always* be placed next to S_{i0}, the corresponding data in snapshot 0.

In the edge array of the time-locality layout, an edge is uniquely identified by the two vertices it connects and all the edges are grouped by their source (or destination) vertices. Each element in the edge array represents an edge. It contains a vertex id to index the corresponding vertex data in the vertex array. It is also associated with a snapshot bitmap specifying the snapshots that contain the edge. For example, the bottom-right of Figure 3 shows an element in the edge array that represents an edge e_{ij} from vertex v_i to v_j. The value of the snapshot bitmap is 110, indicating that the edge exists in snapshots 0 and 1, but not in snapshot 2. The snapshot bitmap saves the memory footprint and provides an efficient way to check whether or not a snapshot contains an edge. Edge e_{ij} may have associated data (e.g., edge weight) in the corresponding snapshots designated by the snapshot bitmap. For example, w_{ij}^0 in Figure 3 denotes the associated data of e_{ij} in snapshot 0.

Although data locality on the graph structure can be captured by carefully placing vertices in a static graph snapshot [23], the real performance gain of this scheme heavily relies on the actual structure of graph. After all, the graph structure is not a linear structure and is hard to be placed on a linear address space with good locality. In contrast, we have found it easier to exploit the data locality on the time dimension because the data access pattern on this dimension are often linear. Chronos therefore favors the time-locality layout, coupled with locality-aware batching scheduling that we describe next.

3.3 Locality Aware Batch Scheduling

Chronos argues for a scheduling mechanism that combines the considerations on both the execution of the operations and the underlying data layout strategy. The scheduling should make the data access pattern aligned with the underlying data layout to achieve better locality. For example, if the in-memory layout places the states associated with a vertex across multiple snapshots together in the time-locality layout, it would be ideal to schedule computation that operates on those states together. If the in-memory layout places a vertex close to its neighbors in the same snapshot in the structure-locality layout, it would be ideal to schedule computation that operates on those vertices together.

Instead of doing computation snapshot by snapshot on the structure-locality layout, LABS aligns the data access pattern of the computation with the time-locality layout. To do this, LABS *batches* the processing on each vertex across all the snapshots. Similarly, for each edge of a vertex, LABS performs the propagation to a neighboring vertex for all the snapshots in a batch. Because the vertex data for all the snapshots are placed contiguously in the time-locality layout, LABS can therefore exploit the excellent data locality in the time dimension.

Besides data locality, this batched scheduling produces another advantage to save the times to enumerate the edge array. LABS only enumerates the edge array *once* for processing all the snapshots; otherwise, the snapshot-by-snapshot scheduling would involve one enumeration for the edge array for each snapshot. This results in fewer memory accesses for LABS.

3.4 Parallel Processing with LABS

LABS can use multiple cores to parallelize graph mining on a series of snapshots. There are two design choices for parallelization. We can either assign each snapshot to a CPU core (which we call *snapshot-parallelism*), or partition each snapshot by vertices and assign each partition to a core, which we call *partition-parallelism*.

For example, assume we have two CPU cores c_0 and c_1, and two snapshots S_0 and S_1. The snapshots can be partitioned into two parts, P_{00} and P_{01} for S_0, and P_{10} and P_{11} for S_1. The snapshots are partitioned in a consistent way such that a vertex that exists on both snapshots is assigned to the partition with the same id. In snapshot-parallelism, we assign S_0 and S_1 to c_0 and c_1, respectively, and let the two cores run concurrently. In partition-parallelism, we assign $\{P_{00}, P_{10}\}$ to c_0 and $\{P_{01}, P_{11}\}$ to c_1.

There are interesting trade-offs between different types of parallelization strategies. Snapshot-parallelism does not involve synchronization among cores because computations on different snapshots are independent. The strategy is however fundamentally incompatible with LABS. In contrast, partition-parallelism incurs the overhead of inter-core communication. It requires locks to protect concurrent vertex data propagation due to the cross-partition edges. Nevertheless, partition-parallelism can be enhanced with LABS for better locality. Meanwhile, with LABS, the inter-core synchronization cost of partition-parallelism can be significantly reduced because the lock on a vertex and the propagation through an edge can be performed in a batch for multiple snapshots. Specifically, assume (v_i, v_j) shown in Figure 3 is a cross-partition edge and it exists in N snapshots (2 in this case). Without LABS, the propagations along the edge for N snapshots might involve N rounds of inter-core communications and N times of locking on the destination vertex (i.e., N locks for N snapshots). While with LABS this only introduces one inter-core communication and one locking for all the N snapshots (i.e., 1 lock for N snapshots). Although the 1-lock-for-N-snapshot looks to introduce larger critical section and decrease the concurrency, the fact that the batched propagation along an edge for multiple snapshots incurs similar number of inter-core communication to the propagation for one snapshot makes them similarly fast. Our evaluation results demonstrate that, when integrated with LABS, partition-parallelism can be significantly more efficient than snapshot-parallelism.

3.5 Incremental Computation with LABS

Incremental computation is another effective approach to optimizing graph computation on a series of snapshots [5]. For example, we compute the single-source shortest path (SSSP) on a graph snapshot S_0 with vertex v_0 as the source vertex. After the computation, each vertex has a computed associated data representing the distance between the vertex and v_0. When we perform the same computation on snapshot S_1 , we use the computed result on S_0 as the initial value for the current computation on S_1. The convergence of the computation can be much faster when the distances between v_0 and most of the vertices do not change in S_1 compared to those in S_0.

However, incremental computation has its limitations. For example, some incremental SSSP algorithms are designed to handle edge insertion only (or edge removal only) [25]. Moreover, reusing results in the previous snapshot does not always reduce the computation time. In an extreme case, one edge removal near v_0 might make it disconnected from the major part of the graph. In this case, most vertices in the graph have to recompute and update their distances to v_0.

Chronos enhances incremental computation in two significant ways. First, to compute N snapshots from S_0 to S_{N-1}, Chronos first computes the result for snapshot S_0. After having the result for S_0, Chronos then computes the rest of $N-1$ snapshots (S_1 to S_{N-1}) in a batch using LABS. In the batch processing, Chronos uses the result computed in snapshot S_0 as the initial value for the $N-1$ subsequent snapshots, thereby enabling incremental computation. While the total amount of computation in this way might be higher than a pure incremental computation approach (which computes on the snapshots in a serial order), our approach does benefit from better locality as well as the reduced number of accesses to the edge array.

Second, because the snapshots are known in advance, for a group of N snapshots, Chronos can pre-compute the *intersection* (or the *union*) of these N snapshots, so that each true snapshot simply adds (or removes) edges/vertices to that intersection (or union) graph to allow incremental computation even when the algorithm only supports edge/vertex insertion (or removal). This enlarges the scope where incremental computation is applicable. For example, consider an initial graph snapshot $S_0 = (V_0, E_0)$. The next snapshot S_1 might have removed edge e_1, while adding many other edges. The initial graph G_0 for incremental computation can be the intersection of S_0 and S_1, which would be $(V_0, E_0 - \{e_1\})$. Thus S_0 and S_1 can be constructed from G_0 by adding edges only.

3.6 Chronos in a Distributed Setting

Although our experiences with real-world large temporal graphs have shown that multiple snapshots can usually fit in memory on a powerful multi-core machine, we have extended Chronos to a distributed environment, where a series of snapshots are partitioned and assigned to different machines, much like the way they are partitioned and assigned to different cores on a multi-core machine. This allows Chronos to handle even larger temporal graphs when needed.

4. Chronos On-Disk Temporal Graph

Chronos assumes the required temporal graph information has already been prepared in the desired in-memory layout before the execution of temporal graph mining. This preparation step relies on how the system stores temporal graphs on disk.

4.1 Data Model of the On-Disk Temporal Graph

Chronos models the "evolution" of a graph and treats a temporal graph as a series of *activities*, where an activity involves the addition, deletion, and modification of vertices, edges, or their associated data at a particular point in time. For example, $\langle \mathsf{delV}, v_6, t_1 \rangle$ is an activity that removes a vertex v_6 at time t_1, $\langle \mathsf{addE}, (v_6, v_1, w), t_2 \rangle$ adds a new edge from v_6 to v_1 with a weight w at time t_2, while $\langle \mathsf{modE}, (v_6, v_1, w'), t_3 \rangle$ modifies the weight associated with edge (v_6, v_1) to w' at time t_3.

One fundamental design choice for Chronos is how to store temporal graphs as they are updated continuously. There is an inherent tradeoff between storing pre-computed graph state (which enables fast queries) and the space it occupies in memory and on disk (or SSD). Specifically, consider a strawman approach that stores every graph update activity in a log. Although such a format is compact and simple to implement, computation on a temporal graph requires expensive reconstruction of all the graph snapshots within the queried time range. Contrast this with an alternate strawman approach where we checkpoint a full graph snapshot whenever it is modified. This approach produces graphs that are easier to query, but introduces too much redundancy.

To create a compact layout without sacrificing performance, Chronos introduces the notion of *snapshot groups*. A snapshot group, G_{t_1,t_2}, consists of the state of graph G in the time range $[t_1, t_2]$. Specifically, it contains a *checkpoint* of the entire graph at the start time t_1 and all graph updates made until t_2. Therefore, a temporal graph consists of a series of snapshot groups of successive time ranges.

A snapshot group G_{t_1,t_2} contains enough information to access the graph snapshot at any time point in the time range, although accessing a snapshot at a time t after t_1 is more expensive because Chronos must read all updates made from t_1 to t and merge them to the checkpoint at t_1. Chronos further allows a user to specify a redundancy ratio (representing the allowed maximum percentage of redundant data) to control the size of the snapshot group.

Depending on applications, a snapshot group is stored as edge files (for edge-related states and activities) and vertex files (for vertex-related ones). For example, there can be one vertex file for the rank values and others for other vertex-associated properties. In the rest of this section, we focus on the description of an edge file because all edge/vertex files are treated the same way.

4.2 Time-Locality Graph Layout

Similar to the in-memory temporal graph structure, Chronos uses the time-locality graph layout for on-disk temporal graphs as well. In order to reconstruct a series of snapshots, Chronos needs to retrieve all activities associated with a vertex v that falls into the time interval of a snapshot group. The time-locality layout is designed for such a query as it groups together all activities associated with a vertex.

Figure 4 illustrates the time-locality layout of an edge file. The file starts with an index to each vertex in this snapshot group, followed by a sequence of segments, each corresponding to a vertex. The index allows Chronos to locate the starting point of a segment corresponding to a specific vertex without a sequential scan. A segment for a vertex v_0 consists of a checkpoint sector C_0, which includes the edges associated with v_0 and their properties at the start time of this snapshot group, followed by edge activities associated with v_0. For example, C_0 might contain information in the form of $(v_0, v_1, w_1), (v_0, v_5, w_2), \ldots, (v_0, v_n, w_m)$, which indicates that the graph snapshot at the start time of snapshot group contains edges (v_0, v_1) with weight w_1, (v_0, v_5) with weight w_2, (v_0, v_n) with weight w_m, and so on. The sequence $(a_{01}, a_{02}, \ldots, a_{0t})$ refers to edge activities related to v_0, sorted in time order where $a_{01} = \langle \mathsf{addE}, (v_0, v_6, w), t_1 \rangle$ adds a new edge (v_0, v_6) with weight w at time t_1, $a_{02} = \langle \mathsf{modE}\ , (v_0, v_1, w'), t_2 \rangle$ changes the weight of edge (v_0, v_1) to w' at time t_2, and $a_{0t} = \langle \mathsf{delE}, (v_0, v_5), t \rangle$ removes edge (v_0, v_5) at time t.

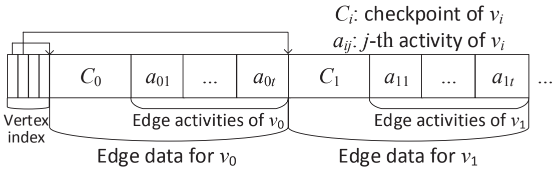


Figure 4. The time-locality format.

To speed up the process on a temporal graph to identify and reconstruct the state associated with a vertex/edge at a particular time t, Chronos further introduces a link structure for each activity a_{ij}. The structure links to the next activity $a_{i(j+1)}$ associated with the same vertex/edge. In practice, Chronos adds a new field t_u in an activity; the value of this field is set to infinity if the activity is the last one in the snapshot group for that edge or vertex. To find the state at time t for a vertex v, Chronos scans the activities in the time order until it hits an activity at t_1 with its t_u field set to t_2, such that $t_1 \leq t < t_u = t_2$ holds, because this is the last activity for this vertex before or at time t.

4.3 Preparing the In-Memory Layout for Chronos

Chronos provides an interface to load a series of snapshots from one or more snapshot groups into memory. The implementation of this interface loads the time-locality on-disk layout to reconstruct the states of the specified snapshots through a sequential scan on the snapshot groups. The on-disk time-locality layout is convenient as it matches the time-locality in-memory layout for Chronos. It is worth noting that the on-disk image contains all the information related to a temporal graph and records the activities faithfully, while the in-memory layout is optimized for the particular graph mining in that it stores the reconstructed *states* rather than update activities (delta) and that it is in a compact format with only the necessary information.

Our experiences show that, when loading the on-disk temporal graph and reconstructing the snapshots, Chronos can always saturate the bandwidth of disk (even the SSD hard drive). In our experiments, the cost of loading on-disk data is often a small fraction of the end-to-end graph computation time.

5. Implementation

As mentioned in Section 2.2, most existing graph engines support the *scatter-gather* iterative graph computation model. The model can be implemented in different modes with subtle differences that could have deep implications on performance. In this section, we introduce different ways to implement the scatter-gather model and note that Chronos, including the key design elements like the time-locality layout and LABS, is fully compatible to these implementations.

The scatter-gather model can be implemented in a *vertex-centric* way [7, 16, 17, 23] or *edge-centric* [27] way. A vertex-centric graph engine iterates over vertices, it requires that users provide a scatter function and a gather function for each vertex. The scatter function specifies the behavior of each vertex where it computes and propagates (scatters) the local value to its neighbors; the gather function dictates the vertex behavior when it gathers updates from its neighbors. A vertex-centric engine uses two versions of the vertex data array (see Figure 3 in Section 3.2) during the computation: one stores the value computed in the previous iteration, and the other keeps the most updated value computed in the current iteration. The two vertex data arrays switch the roles after each iteration.

An edge-centric engine like X-Stream [27] iterates over edges rather than vertices. It requires not a vertex associated function but, for each edge, an edge_scatter function and an edge_gather function that describe what value needs to be propagated through the edge and where the value associated with the edge needs to be applied, respectively. In the scatter phase, the edge-centric engine scans the edge array and writes the data computed from source vertices to an update array *sequentially*; in the gather phase, it *sequentially* reads the computed data stored in the update array and applies to the destination vertices. In order to make scatter and gather operations as sequential as possible, the edge-centric engine introduces an additional *shuffle* phase between the scatter and gather phases to partition the update array by the destination vertex. The similar shuffle operation is also taken on the edge array by the source vertex, before the computation starts. This way, the engine can mitigate the random data accesses by streaming the updates with edges into and out of sequential buffers (i.e., update edge array). This graph-computation mode maximizes the chances of sequential data processing and is referred to as the *stream* mode.

A vertex-centric engine can operate in the *push* mode or the *pull* mode. In the push mode [17, 23], the engine checks the change of the vertex state from the previous iteration. It sends the value to all the neighbors for updating only when there is a change that is significant enough. In the pull mode [5, 16, 29], on the other hand, the engine collects the states of the neighbors of a vertex by pulling, rather than pushing the changes. Although seemingly similar, the push and pull modes have important differences that have significant performance implications. For example, consider the execution on a single multi-core machine, where the computation on each vertex can be executed concurrently. In the push mode, a vertex needs to use a lock to protect the process of updating the value of a neighbor because multiple vertices (with an outgoing edge to the same vertex) might be updating the state concurrently. In contrast, in the pull mode, no locks are needed because, in each iteration, a vertex only needs to read the value in the previous iteration from its neighbors. This value is stored in the aforementioned two-version vertex data array and is immutable during the current iteration. Moreover, in the pull mode a vertex is the only entity reading and updating its own state in the current iteration. Although no lock is required, the pull mode needs to pay the cost to check the significance of the value changes of the neighbors. This checking on a neighboring vertex needs to be performed multiple times if different vertices share a same neighbor, which is a significant overhead that sometimes overweights the saving of locks (see details in Section 6).

Regardless of the different implementations, being vertex-centric or edge-centric model, push, pull, or stream mode, Chronos together with LABS is beneficial to all different graph-engine implementations. We have implemented Chronos with all the previously described implementations. We demonstrate the effectiveness of all the implementation and explain the detailed reasons in Section 6.

6. Evaluation

We evaluate Chronos both on a commodity multi-core machine and on a small distributed testbed. The multi-core machine is equipped with dual $2.4\,\mathrm{GHz}$ Intel Xeon E5-2665 processors (16 cores in total), $128\,\mathrm{GB}$ of memory. The distributed testbed consists of 4 multi-core servers with the same configuration and interconnects with InfiniBand (double data rate, $40\,\mathrm{Gbps}$).

We conduct the experiments using 5 graph applications and 4 large real world temporal graphs. The selected applications include PageRank [3], weakly connected component (WCC), single-source shortest path (SSSP), maximal independent set (MIS), and sparse matrix-vector multiplication (SpMV). Each of them is computed on a series of snapshots of different temporal graphs. We use the following 4 real world temporal graphs in the experiments.

Wikipedia reference graph (Wiki): This is a temporal graph of English Wikipedia web pages. Each Wiki page is a vertex. An edge activity $\langle (m, n), t \rangle$ represents a hyperlink from page m to n created at time t [18]. The Wiki graph consists of 1.87 million vertices and 40 million hyperlinks. It shows how the English Wiki network evolved over a period of 6 years.

Time-evolving web graph (Web): The graph is similar to the Wiki graph except that each vertex now is a webpage in the .uk domain and an edge denotes a link between pages at a certain time instance. The temporal web graph includes 12 monthly snapshots in the .uk domain [2]. Each edge activity is associated with the creation or removal time observed in the snapshot. In total, the web graph contains more than 133 million vertices and 5 billion edges.

Twitter mention graph (Twitter): In Twitter, a tweet containing a string like “@tom” means that the publisher mentions user “tom”. In a Twitter mention graph, a user is a vertex. An edge activity $\langle (m, n), t \rangle$ in the mention graph means that user m mentioned n in a tweet at time t. The mention graph indicates the amount of attention each user pays to others and how the attention changes over time [26].

Graph	# of vertices	# of edge activities	Time span
Wiki	1.871 M	39.953 M	6 Y
Twitter	7.512 M	61.633 M	3 Mon
Weibo	27.707 M	4.900 B	3 Y
Web	133.633 M	5.508 B	12 Mon

Table 1. Temporal graph statistics (M: million, B: billion, Y: year, Mon: month).

We have collected more than 102 million tweets over three months, from which we derived a temporal graph with around 7 million vertices and 61 million edge updates (events).

Weibo mention graph (Weibo): Weibo [30] is the Chinese counterpart of Twitter. The meaning of the Weibo mention graph is the same as that of Twitter. We have collected more than 7 billion Weibo microblogs over three years. The derived temporal graph contains around 28 million vertices and 4.9 billion edge updates.

The size and the time span of each temporal graph are summarized in Table 1.

For parallel/distributed graph computation, we partition the graphs using Metis [8], a public implementation of multilevel k-way graph partitioning algorithm. Metis is believed to be an effective partitioning method for a general graph to minimize cross-partition edges and balance among partitions. Within each partition, we use spectral placement to order the vertices in the graph layout for better locality on the graph-structure dimension [23]. Note that it is believed that to partition the graph by *edge*, which may cut a single vertex into multiple replicas across partitions, is an effective way to partition graphs exhibiting power law [4, 7]. All these partitioning techniques are compatible and complimentary to Chronos. And a particular graph partitioning technique alone does not affect the performance advantage of LABS if under the same configuration.

6.1 Effectiveness of LABS

To demonstrate the advantage of the proposed temporal graph layout and LABS, we first study the performance of Chronos in the single-thread case. We use the straightforward approach of running graph computation on each snapshot one by one as the baseline for our comparisons. To generate N snapshots for our experiments, we equally divide the second half of the entire time range by N to have snapshots covering non-overlapping time ranges. The first snapshot is chosen in the middle of the entire time range to generate a graph large enough that is meaningful for large-scale graph computation. An important parameter for LABS is its *batch size*, which is the number of snapshots that are batched together for iterative computation in LABS. Note that our baseline is essentially the execution with batch size set to 1.

Figure 5 shows the performance of Chronos compared to our baseline, for different applications on the Wiki, Twitter, and Weibo graphs. We compare the performance in all three processing modes: push, pull, and stream. As the figures show, Chronos outperforms our baseline consistently across all applications and in all the three modes. The gain becomes more significant when the batch size increases. When the batch size is 32, Chronos runs more than 20 times faster for SSSP on Weibo graph in the pull mode.

Our further investigations on cache misses and edge accesses show that locality and batching across snapshots are the underlying reasons for Chronos' superior performance and also help explain some of the differences observed in different modes. We report those numbers next.

Batch size	L1d	LLC	dTLB
Push mode			
1	8,759	649	3,462
4	3,865	584	1,003
16	1,107	265	287
32	687	196	160
Pull mode			
1	6,470	859	3,419
4	2,638	753	839
16	926	365	230
32	635	272	126
Stream mode			
1	4,091	1,090	79
4	1,290	274	23
16	493	95	10
32	386	62	9

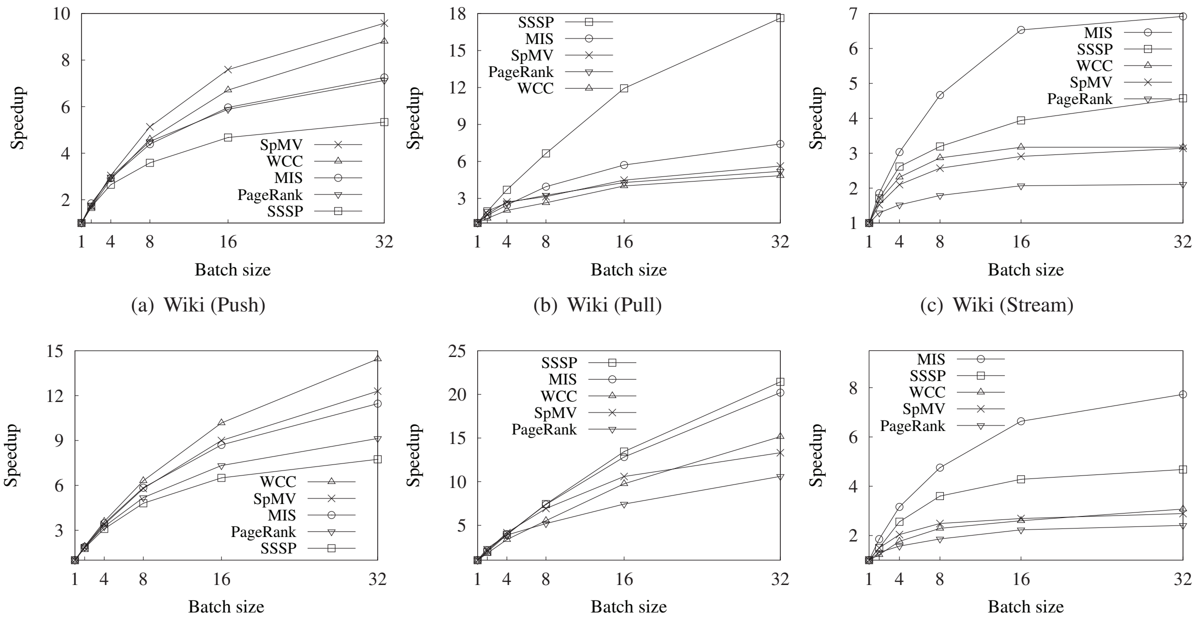
Table 2. CPU L1d, LLC and dTLB miss counts in three modes for MIS on Wiki graph (in millions). L1d: level 1 data cache, LLC: last level cache, dTLB: data translation lookaside buffer.

Reduced cache-miss counts. Table 2 shows the numbers of CPU cache misses and TLB misses for MIS (one iteration) on the Wiki graph; we have observed similar effects for other applications on other graphs (omitted). The numbers are measured through hardware CPU performance counters. As shown, the miss count decreases with the increase of the batch size. This suggests that the speedup is due to better data locality and explains why a larger batch size brings more gains.

In the push mode, Chronos enables consecutive writes for multiple snapshots, which reduces the number of cache misses. Likewise, the consecutive reads in the pull mode brings similar benefits to Chronos. In the stream mode, Chronos is particularly beneficial in the shuffle stage [27]. The shuffling of the edge-associated values in multiple snapshots is performed consecutively in a batch, thereby reducing the number of cache misses.

Note that in the stream mode the TLB-miss count is small compared to other modes due to its streaming behavior. Also, even when the batch size is 1, the stream mode reduces the chance of random access, which in turn reduces the cache miss count. This explains why we observe the least gain in the stream mode.

Reduced access to edge array. Another factor contributing to Chronos' better performance is the batching effect across



(a) Wiki (Push)

(b) Wiki (Pull)

(c) Wiki (Stream)

(d) Weibo (Push)

(e) Weibo (Pull)

(f) Twitter (Stream)

Figure 5. Chronos single-thread speedup in computation time.

Graph	BS: 1	BS: 4	BS: 16	BS: 32
Wiki	757 M	200 M	62 M	40 M
Twitter	1193 M	323 M	104 M	62 M

Table 3. Number of edge array access for PageRank in the first iteration. BS: batch size.

snapshots, which reduces the number of edge-array accesses. For each propagation, a vertex needs to access its edges in the edge array to find out its neighbors before it can propagate the value in the push and stream mode, or pull the value in the pull mode. In Chronos, the edge access is done for all batched snapshots once, rather than once for each snapshot. Larger N brings larger benefits due to more saved accesses to edge array. Table 3 shows the numbers of accesses to edge array for PageRank in the first iteration on the Wiki and Twitter graphs. In the first iteration, the number of edge accesses in push, pull, and stream modes is the same. As expected, the larger the batch size, the fewer the number of edge accesses in the edge array.

Chronos with incremental computation. We then show the benefit of LABS-enhanced incremental computation in Chronos compared to the standard incremental computation approach. In the experiment, we compute 128 snapshots that are evenly spread over the last 10 months of the Wiki graph (from June 2006 to March 2007). Two adjacent snapshots are separated more than 2 days apart, which account for more than 130k edge activities on average.

The standard incremental computation approach runs on each snapshot in sequence, using the result of the previous snapshot. The LABS-enhanced incremental computation with a batch size of n firstly computes the first snapshot S_0 and incrementally computes the next n snapshots $(S_1 ... S_n)$ using LABS by reusing the result of S_0. It further computes the following n snapshots $(S_{n+1} ... S_{2n})$ incrementally by reusing the result of the snapshot S_n. The computation moves forward until all the 128 snapshots are calculated.

Figure 6 shows the comparisons for WCC and SSSP on the Wiki graph in the single-thread case. We choose to use the push mode because in this mode only updates are propagated, making incremental computation more effective. The x-axis represents different batch size in LABS, and the y-axis shows the performance improvement of the proposal in percentage. Note that the case where batch size equals to 1 is the standard incremental computation.

The figure shows that our proposal can outperform the naïve incremental method more than 60%. Initially, the increase of the batch size brings more benefits due to a larger batching effect as previously explained. When batch size becomes even larger, the difference between later snapshots (e.g., $S_1 ... S_n$) and the initial snapshot (e.g., S_0) is also larger. This introduces more *duplicated* computation for later snapshots, assuming a simple approximate model where the amount of incremental computation between two snapshots is proportional to the number of changes. For example, to calculate S_n from S_0 incurs more duplicated computation than to compute S_n from S_{n-2} (if $n > 2$). Hence when the batch size is large enough, such unnecessary computation results in a reduced performance gain, as shown in Figure 6. A system should strike a balance between batching effects and the incremental computation.

Note that in the multi-thread case, the benefit of LABS-enhanced incremental computation will be amplified due to

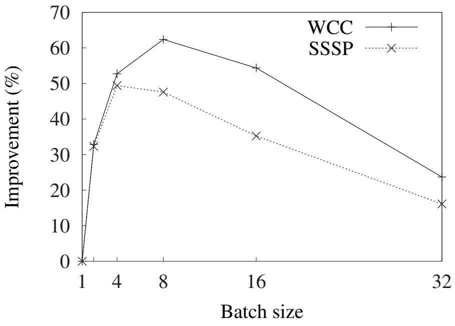


Figure 6. The performance gain of incremental LABS against standard incremental computation with varying batch size on Wiki graph.

other advantages in multi-core settings, as will be discussed in Section 6.2.

6.2 Chronos Performance on Multi-Core Machines

The performance of temporal iterative graph mining on a multi-core server is heavily influenced by the subtle interplay among data locality, inter-core communication, and other factors such as lock contentions. Our evaluation has shown that Chronos continues to outperform alternative designs and its advantage is sometimes even amplified.

We compare Chronos to two recent in-memory graph engines, Grace [23] and X-Stream [27], which are both optimized for multi-core machines. Grace is a vertex centric graph engine. The original implementation of Grace only supports the push mode, we further extend Grace to support the pull mode. X-Stream is edge centric graph engine that supports the stream mode. We modify X-Stream to support snapshot-parallelism. It is worth pointing out that the key design of Chronos can be integrated with different existing graph engines such as Grace and X-Stream, making it widely applicable and useful in enhancing existing graph engines.

In the experiment, we compute 32 snapshots with an equal time-interval across the second half of the entire time range. (We select the first snapshot at the middle of the time range to have a snapshot large enough for a meaningful parallel computation.) All experiments use a batch size of 32. Figure 7 shows the results on the Wiki graph as the computation uses different numbers of cores. We are using the same baseline as in the single-threaded case. Partition-parallelism and snapshot-parallelism are used in this set of experiments; The y-axis denotes the speedup compared to our baseline. Note that, even on a single core, Chronos with batch size 32 has already achieved significant speedups, as shown in the previous experiment (Figure 5). We are seeing more than 10 times of an additional speedup on 16 cores (without hyperthreading). As shown in Figure 7, Chronos scales better than Grace and X-Stream in all the three modes and for all the applications. We observe similar effects in other graphs (e.g., the Weibo/Twitter graph in Figure 8) and other applications. We discuss the differences in three modes that lead to different performance behaviors at the end of this section. Next we report the results of in-depth analyses that help explain the performance advantages of Chronos. We further discuss the results of snapshot-parallelism later on.

	Push			Pull		
# of cores	2	4	8	2	4	8
Chronos	23.1	58.6	105.2	31.0	55.8	71.5
Grace	977.6	2471.6	4244.2	1740.4	3047.9	3923.8

Table 4. The number of inter-core communications for PageRank on the Wiki graph (in millions).

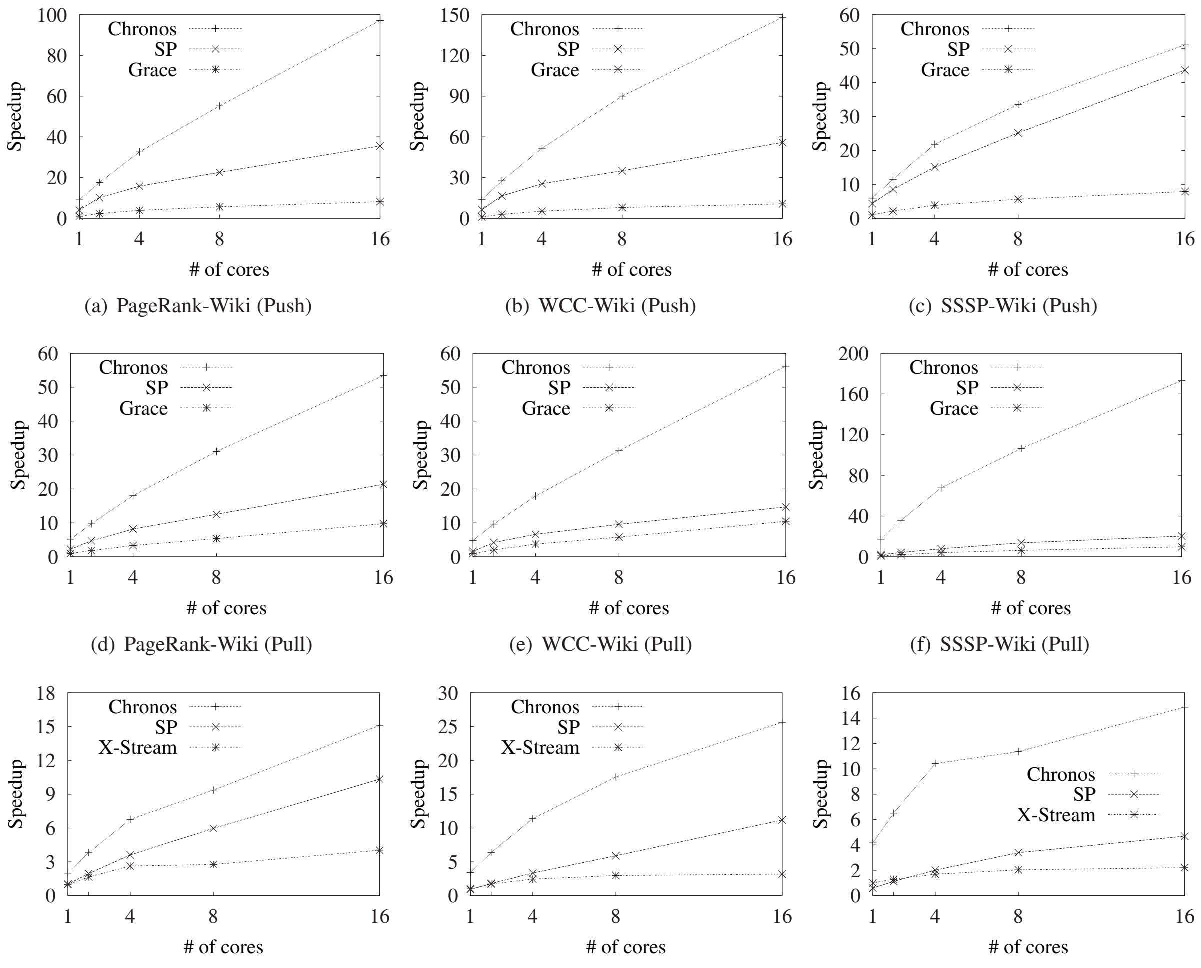
Reduced inter-core communications. Our further investigation reveals that one key reason that Chronos outperforms Grace is the reduced inter-core communication cost. In the pull mode, Chronos pulls updates of a vertex from a remote core. Chronos performs such remote reads in a batch (across multiple snapshots). Because the vertex values in consecutive snapshots are placed together, Chronos reduces the number of remote reads: values of multiple snapshots are likely stored within a cache line. The push mode in Chronos has the same benefit for a similar reason except that a remote read becomes a remote write (i.e., push). In the stream mode, the inter-core communication is not a dominant factor because each CPU core mainly communicates with the memory. The data exchange between cores are done using memory indirectly.

Table 4 shows the inter-core communication overheads in the push and pull mode for PageRank (one iteration) on the Wiki graph. As the table shows, the number of inter-core communications (measured through hardware performance counter) of Chronos is significantly smaller than that of Grace in various multi-core settings.

Reduced lock contentions in the push mode. In the push mode, when a vertex performs a write operation to another vertex, it needs to acquire a lock to the destination vertex because multiple vertices may write to the same destination concurrently. This is another source of overhead in the multi-core setting. LABS manages to reduce such lock contentions in the push mode for Chronos as it acquires locks in a batch across snapshots.

Table 5 shows the total spinlock running time, an indication of the level of lock contention, in the push mode for PageRank on the Wiki graph (1 iteration). It shows that Chronos incurs one order of magnitude fewer contentions than Grace. Similar trends can be observed for other applications in the push mode. Note that lock contention is not a critical issue in the pull and stream modes.

Snapshot-parallelism. Temporal graphs provide more choices for parallel computation. Snapshot-parallelism assigns each snapshot for one CPU core to compute. There is no lock contention or inter-core communication. However, the computation within each CPU core cannot exploit lo-



(a) PageRank-Wiki (Push)

(b) WCC-Wiki (Push)

(c) SSSP-Wiki (Push)

(d) PageRank-Wiki (Pull)

(e) WCC-Wiki (Pull)

(f) SSSP-Wiki (Pull)

(g) PageRank-Wiki (Stream)

(h) WCC-Wiki (Stream)

(i) SSSP-Wiki (Stream)

Figure 7. Performance comparisons on multi-core with the Wiki graph. SP: snapshot-parallelism.

# of cores	2	4	8	16
Chronos	1.32s	1.34s	1.85s	4.02s
Grace	28.85s	34.25s	47.54s	96.73s

Table 5. The level of lock contention comparison for PageRank on the Wiki graph. s: second.

cality to reduce cache misses as the LABS mechanism in Chronos does. Even for snapshot-parallelism, we use the same in-memory layout as described in Section 3.2; in particular, there is a single read-only edge array shared by all snapshots; the edge array uses the snapshot bitmap for compression. All cores will access the same edge array during computation, but no locking is needed as the array is read-only. This format reduces the in-memory footprint and can potentially even reduce cache misses. However, snapshot-parallelism cannot benefit from the reduced access to the edge array, as LABS does.

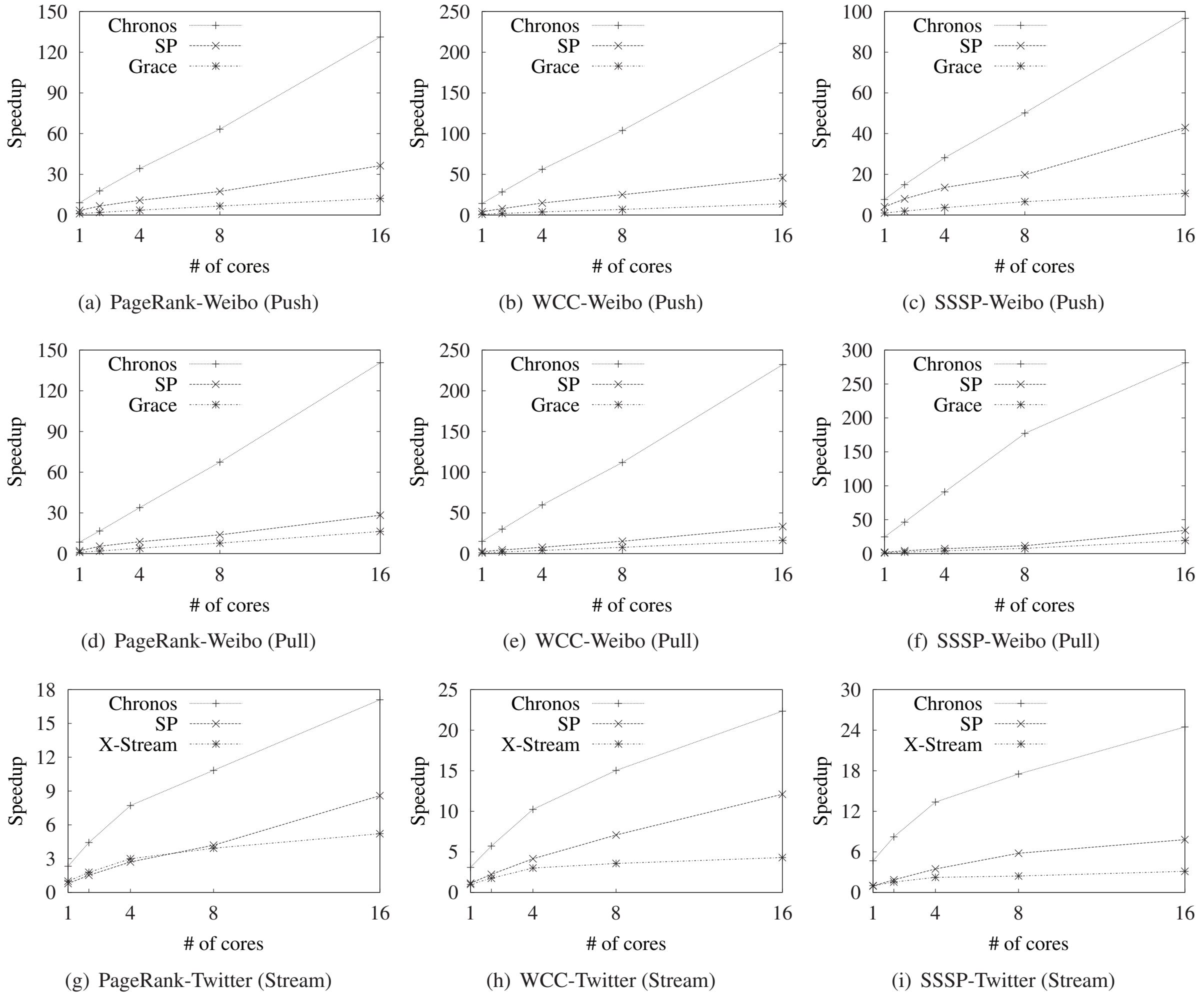
Figures 7 and 8 show the performance of various applications in different mode on different temporal graph for snapshot-parallelism and Chronos. It shows that the performance of snapshot-parallelism is worse than Chronos. We observe similar trends for other applications.

Note that in the stream mode, snapshot-parallelism is sometimes slower than X-Stream when the degree of parallelism is low. This is because, in order to support snapshot-parallelism, we use some auxiliary data structure to extend the implementation of X-Stream, which increases the randomness of memory access.

Snapshot-parallelism in all the three modes is able to consistently outperform Grace or X-Stream with the increase of core due to better parallelism (e.g., fewer inter-core communications).

Other observations. Finally, we briefly comment on the difference of the push, pull, and stream modes.

As explained, the push mode requires heavy locks for data propagation. The pull mode, on the other hand, reads data from other vertices concurrently and does not require locks. The stream mode is nearly lock-free: it only requires a few lightweight atomic operations in the scatter phase [27].



(a) PageRank-Weibo (Push)

(b) WCC-Weibo (Push)

(c) SSSP-Weibo (Push)

(d) PageRank-Weibo (Pull)

(e) WCC-Weibo (Pull)

(f) SSSP-Weibo (Pull)

(g) PageRank-Twitter (Stream)

(h) WCC-Twitter (Stream)

(i) SSSP-Twitter (Stream)

Figure 8. Performance comparisons on multi-core with the Weibo and Twitter graphs. SP: snapshot-parallelism.

In the pull mode, in order to detect whether the value of neighbors has been updated a vertex has to check the "dirty" bit of neighbors. Each vertex has to scan the edge array to find out its neighbors, thus requiring $O(|E|)$ access, where E is the set of edges in the graph. In contrast, in the push mode each vertex needs to check the dirty bit of its own only, there resulting in $O(|V|)$ access, where V is the set of vertices in a graph. Because $|E|$ is typically significantly larger than $|V|$, this cost is higher in the pull mode than that in the push mode. In addition, the overhead of a read operation to check the dirty bit of remote vertices are more significant in multi-core environments especially when neighbors are located in another CPU core, leading to even higher overhead. Experiments show that this can result in worse performance in the pull mode for some applications such as SSSP, than that in the push mode.

In our experiments, we have observed that the memory footprint in stream mode is significantly larger than in the other two modes due to its edge-centric nature. In fact X-Stream cannot accommodate the Weibo graph on a single machine with 128GB memory (note that X-Stream is capable of leveraging external memory, which is out of the scope of this paper). Moreover, unlike in the push and pull modes where updates go directly to the destination vertex, the stream mode achieves this indirectly through edges and has an extra shuffling stage. This incurs additional read/write operations.

Our experiences with the three modes indicate that no single mode is the best for all applications on all graphs, due to the different tradeoffs in each mode. However, despite the different performance characteristics in different modes for different applications, the benefits of Chronos have shown up consistently in all cases.

6.3 Chronos Distributed Performance

We have set up a small distributed testbed to test whether the benefits of Chronos extend to a distributed setting. In particular, our distributed testbed consists of 4 servers that are connected through InfiniBand. To fully exploit the capability of InfiniBand, Chronos uses MPI for the inter-machine communications.

	Web graph			Weibo graph		
Applications	PageRank	WCC	SSSP	PageRank	WCC	SSSP
Chronos	472s	332s	124s	2002s	1250s	48s
Baseline	781s	670s	136s	7318s	6405s	518s

Table 6. Chronos performance in distributed environments. Baseline: to compute snapshot by snapshot.

Table 6 shows the running time for different applications (5 iterations) on the Web and Weibo graphs in the push mode. The web graph has 12 snapshots, so we set the batch size to 12. The Weibo graph runs on 32 snapshots. To focus more on the distributed environment, we use a single thread on each server. The results show that Chronos can run more than 3 times faster than the naïve implementation that computes snapshot by snapshot (PageRank on Weibo). Note that applications run slower in the Weibo graph because the number of cross-partition edges is much larger than that of the web graph: the ratio between inter-partition and intra-partition edge number is 3:1 in the Weibo graph and 1:2 in the web graph.

Because network communication incurs high overhead, the gains from better locality in Chronos are smaller in the end-to-end performance, compared to a single-machine setting. We expect the benefit to be less visible in a more network-constrained environment, where the network communication cost dominates, even though our solution does enable batching across snapshots to make communication more effective.

7. Related Work

Chronos enables efficient temporal iterative graph mining. The key technique that differentiates Chronos from existing graph engines is the joint design of temporal graph layout and the scheduling mechanism (LABS) to fully exploit data locality of temporal graphs and batching effect.

The importance of data locality is well known [6] and has been studied in the context of multi-core graph engines in Grace [23] and X-Stream [27]. Chronos further advocates to exploit data locality along the time dimension, even at the expense of trading data locality in graph structure.

Chronos is complementary to the recent research on graph engines in that it is applicable not only to the vertex centric graph engines, whether it is push based [17, 23] or pull based [5, 16, 29], but also to those with edge centric, stream based graph engines [7, 11, 27]. Chronos further explores the interactions with techniques such as incremental computation [5, 19] to understand the tradeoffs between incremental computation and locality.

The proposed temporal graph data layout and LABS scheduling are also effective in distributed environments. It explores an orthogonal dimension in the design space and is largely complementary to techniques such as dynamic load balancing, priority scheduling, automatic pull/push mode switching, and fine-grained synchronization [9, 20, 29, 31].

Mining temporal graphs has uncovered important properties in real world temporal graphs [1, 13, 32]. More recently, Ren *et al.* [24] study the computation of shortest path on a series of snapshots in a temporal graph. Khurana *et al.* [10] study efficient ways to retrieve a certain or several snapshots of a temporal graph.

Temporal data query has been studied extensively in the relational data model. Salzberg *et al.* surveyed the access methods for time evolving data in [28]. In a relational data model, historical data access can be characterized as key/time based point query (i.e., given a specific key and time) or range query (i.e., both the key and time can be a range). A variety of tree-based index like R-Tree, Time Split B-Tree (TSB tree) [15] and HV-Tree [35] has been proposed for the key/time based queries [28]. Several database systems like the TSB-tree based ImmortalDB [14] and PostgreSQL [21] support such key/time based data lookup.

For temporal iterative graph mining, such key/time based query remains useful. For example, queries for a vertex/edge at a given time instance can leverage the techniques for key/time based lookup. Complementing to the key/time based historical data access techniques in relational model, Chronos is optimized for iterative computation in a temporal graph.

There exist other types of iterative in-memory data process engines like Piccolo, Spark, and Naiad [19, 22, 34]. These engines are not specifically designed for graph mining and hence do not consider graph-aware optimizations.

8. Conclusion

Temporal graphs represent an emerging class of applications, which imposes a unique set of challenges that are not being sufficiently addressed by the current systems. A temporal graph has both a spatial dimension and a temporal dimension, which is the source of many design challenges, but also enlarges the design space to offer interesting opportunities beyond what is possible for a static graph. Chronos' locality-aware batch scheduling demonstrates one such opportunity. We believe temporal graphs will become even more important over time and we hope Chronos can inspire further system research in this new area.

Acknowledgments

We sincerely thank the anonymous reviewers and our shepherd Alan Mislove for their valuable comments and suggestions. This work has been partially supported by the National High-Tech Research and Development Plan (863 Project) 2012AA010903, as well as the National Science Foundation for Distinguished Young Scholars of China (Grant No. 61325010).

References

[1] C. C. Aggarwal and H. Wang, editors. *Managing and Mining Graph Data*, volume 40 of *Advances in Database Systems*.

Springer, 2010.

[2] P. Boldi, M. Santini, and S. Vigna. A large time-aware web graph. *SIGIR Forum*, 42(2):33–38, 2008.

[3] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

[4] R. Chen, J. Shi, Y. Chen, H. Guan, B. Zang, and H. Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. Technical Report IPADSTR-2013-001, Shanghai Jiao Tong Univ., 2013.

[5] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *EuroSys*, pages 85–98. ACM, 2012.

[6] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS*, 1999.

[7] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[8] G. Karypis and V. Kumar. METIS - unstructured graph partitioning and sparse matrix ordering system, ver 2.0. Technical report, Univ. of Minnesota, 1995.

[9] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *EuroSys*, pages 169–182, 2013.

[10] U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. In *ICDE*, 2013.

[11] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: large-scale graph computation on just a PC. In *OSDI*, volume 8, pages 31–46, 2012.

[12] K. Lerman, R. Ghosh, and J. H. Kang. Centrality metric for dynamic networks. In *MLG*, pages 70–77, 2010.

[13] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *KDD*, pages 177–187, 2005.

[14] D. Lomet, R. Barga, M. F. Mokbel, G. Shegalov, R. Wang, and Y. Zhu. Immortal db: transaction time support for SQL server. In *SIGMOD*, 2005.

[15] D. Lomet and B. Salzberg. Access methods for multiversion data. In *SIGMOD*, pages 315–324, 1989.

[16] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *PVLDB*, 5(8):716–727, Apr. 2012.

[17] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[18] A. Mislove. *Online social networks: measurement, analysis, and applications to distributed information systems*. PhD thesis, Rice University, 2009.

[19] D. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *SOSP*, pages 439–455, 2013.

[20] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, pages 456–471, 2013.

[21] PostgreSQL. PostgreSQL, 2013. `http://postgresql.org`.

[22] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, pages 1–14, 2010.

[23] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan. Managing large graphs on multi-cores with graph awareness. In *USENIX ATC*, volume 12, 2012.

[24] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng. On querying historical evolving graph sequences. *PVLDB*, 4(11):726–737, 2011.

[25] L. Roditty and U. Zwick. On dynamic shortest paths problems. In *ESA*, pages 580–591, 2004.

[26] D. M. Romero, B. Meeder, and J. Kleinberg. Differences in the mechanics of information diffusion across topics: idioms, political hashtags, and complex contagion on twitter. In *WWW*, pages 695–704, 2011.

[27] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488, 2013.

[28] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Comput. Surv.*, 31(2):158–221, June 1999.

[29] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *PPoPP*, pages 135–146, 2013.

[30] Sina. Weibo, 2013. `http://weibo.com`.

[31] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. Schreiber. Presto: distributed machine learning and graph processing with sparse matrices. In *EuroSys*, pages 197–210, 2013.

[32] C. Wilson, B. Boe, R. Sala, K. P. N. Puttaswamy, and B. Y. Zhao. User interactions in social networks and their implications. In *EuroSys*, pages 205–218, 2009.

[33] L. Yang, L. Qi, Y.-P. Zhao, B. Gao, and T.-Y. Liu. Link analysis using time series of web graphs. In *CIKM*, pages 1011–1014, 2007.

[34] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 2–2, 2012.

[35] R. Zhang and M. Stradling. The HV-tree: a memory hierarchy aware version index. *PVLDB*, 3(1-2):397–408, Sept. 2010.