Optimizing the Interval-centric Distributed Computing Model for Temporal Graph Algorithms

Anonymous Author(s)
Submission Id: 513

Abstract

Temporal graphs assign lifespans to their vertices, edges and attributes. Large temporal graphs are common for finding the shortest paths in transit networks and contact tracing for COVID-19. Programming abstractions like Interval-centric Computing Model (ICM) extend Google's Pregel model to intuitively compose and execute time-dependent graph algorithms in a distributed environment. However, the *TimeWarp* shuffle and *scatter* messaging in ICM pose performance bottlenecks. Here, we design several optimizations to interval-centric computing. We locally unroll (LU) messages and globally partition the interval graph into windows (WICM) to reduce the time-complexity of TimeWarp. We also defer the execution of the message scatter phase (DS) to reduce redundant messages. We offer a proof of equivalence between ICM and these techniques. Our detailed empirical evaluation for six real-world graphs with up to $133M$ vertices, $5.5B$ edges and 365 time-points, for six temporal traversal algorithms executing on a commodity cluster with 8 nodes, shows that LU, DS and WICM together significantly reduce the average algorithm runtime by $\approx 61\%$ (≈ 15 *mins*) over ICM, and reduce message communication by $\approx 38\% (\approx 3.2B)$ on average.

1 Introduction

Graphs with temporal characteristics are increasingly becoming prominent with applications in financial transactions [13], transportation networks [9] and epidemiology. The vertex and edge structure and the attributes of *temporal graphs* are annotated with a lifespan, allowing vertices and edges to be added and removed; edge weights and vertex labels can have different values during different time intervals. E.g., in a COVID-19 contact graph collected from Bluetooth-based contact tracing apps [3, 34], vertices are users, and an edge exists between two vertices for the interval during which the users were proximate. The edge properties record the time-varying signal strength within the edge's lifespan. Such graphs can grow to millions of vertices, billions of edges, and have months of data.

Time-dependent algorithms, such as temporal reachability and shortest paths, are designed over such graphs after they are collected and materialized [10, 43]. Reachability can help identify individuals required to be quarantined in a COVID-19 contact graph if a *temporally reachable path* from a "source" user who tested positive to another user exists. The time of contact monotonically increases along the path [34]. Similarly, temporal shortest paths are used for planning travel over a road network with time-varying traffic [9, 43].

Google's Pregel provides a distributed programming abstraction to design algorithms over large graphs using a *Vertex-centric Computing Model (VCM)* [25]. These are executed iteratively using a bulk-synchronous parallel (BSP) model [40] through data-parallel vertex computations that update the local vertex state, and message passing at synchronization barriers over a series of supersteps. Some works have extended VCM to design temporal graph algorithms [12, 23, 32] by operating over a series of graph snapshots. However, their vertex-compute costs and messaging overheads can be significantly compounded by the lifespan of the temporal vertices and edges.

Recent approaches [10] mitigate this by operating over a vertex's state intervals and passing messages with lifespans. This attempts to avoid redundant computation and communication over vertex-states and messages that are common for adjacent timepoints. However, such an *Interval-centric Computing Model (ICM)*, in attempting to ease the design of temporal algorithms, introduces a *TimeWarp* phase at each superstep to partition and align the incoming messages with vertex states. The *TimeWarp* operator is super-linear in time complexity with the number of messages received at a vertex. Further, in pipelining the compute and *scatter* communication phases, it may create stale interval messages that are subsumed by a future compute operation, or redundant ones due to interval fragmenting.

With the growing sizes of temporal graphs, there is a strong incentive to enhance the runtime performance for temporal graph algorithms. E.g., finding the temporal reachability from a source vertex for the Twitter interval graph with $52.5M$ vertices, $1.9B$ edges and 30 timepoints takes over 6 *mins* using ICM on a cluster of 8 machines (see Section 8, TR on Twitter). This is despite ICM itself reporting substantially faster performance than other comparable temporal graph platforms [10]. Of this, 5 *mins* are spent in executing *TimeWarp*, and 3.8 B interval messages are generated.

Prior works have enhanced the execution model for Pregel by using a barrier-less asynchronous execution [11], out-of-core computation by spilling vertex states and messages to disk from memory [46], and through vertex migration to balance the load [20]. While such techniques are applicable for processing temporal graph algorithms, they do not directly address the *TimeWarp* overheads or the creation of redundant interval messages by scatter, which are unique to the interval-centric variant of Pregel. Asynchronous execution may increase the number of messages communicated and does not give equivalence guarantees with ICM.

In this article, we propose several novel techniques to accelerate the execution model of interval-centric computing: (1) *Unrolling the temporal intervals* processed in the TimeWarp phase to reduce its time complexity, and (2) *Deferring the scatter phase (DS)* for message creation into a lazy evaluation that reduces redundant calls to messaging. The former is applied both at individual supersteps – *local unrolling (LU)* of messages before computing on an interval vertex, as well as by a global partitioning of the temporal graph into windowed graphs for execution, called *Windowed ICM (WICM)*.

These optimizations do not require any change to the graph algorithms designed using ICM. Further, we also prove that changing the execution model using these techniques still guarantees their equivalence with the results obtained from the ICM execution for a large class of temporal traversal algorithms. Our results for six large temporal graphs using six graph algorithms on an 8-node commodity cluster show that the LU and DS optimizations reduce the runtime of ICM by an average of $\approx 56\%$; Windowed ICM reduces the runtime by $\approx 48\%$ on average over native interval-centric computing, and combining these techniques offers an average reduction of $\approx 61\%$ E.g., this translates to a sizable drop in makespan for the Twitter graph from ≈ 780 *secs* to ≈ 312 *secs* on average.

We make these specific contributions in this paper:

1. We propose three complementary techniques to accelerate the execution model of interval-centric temporal graph algorithms: Local Unrolling (LU), Deferred Scatter (DS) and Windowed ICM (Sections 4, 5 and 6),
2. We offer a proof of equivalence of the proposed optimizations with ICM for a broad class of distributed temporal graph algorithms (Section 7).
3. We perform detailed experiments using diverse real-world graphs and traversal algorithms to evaluate the improvements offered by LU, DS and WICM over ICM, and correlate our analytical model with empirical results (Section 8).

Besides these, we provide a background of interval-centric computing (Section 2), motivate the performance bottlenecks of ICM (Section 3), and review related work (Section 9).

2 Background

A temporal graph is one where entities of the graph: vertices, edges and their properties, have an associated time for which they are valid. An *interval graph* is a temporal graph representation [16] where every entity has a corresponding *time interval* of existence, with a starting timepoint and an ending timepoint that defines its *lifespan*. Formally, an *interval graph* is given by $G = (V, E)$, where V is a set of *interval vertices* $v = (vid, \tau_v)$, each with a unique ID *vid* and its lifespan of existence $\tau_v = [t_s, t_e)$ with start timepoint t_s (inclusive) and

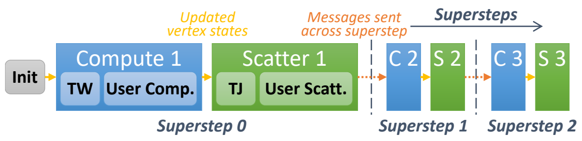
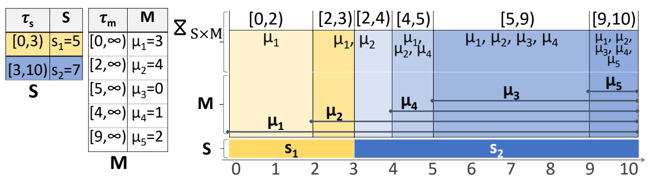
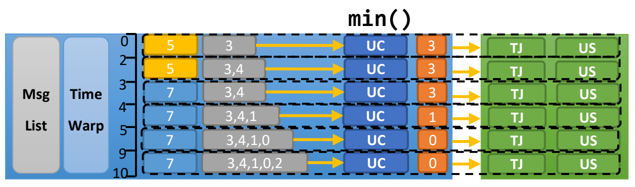


Figure 1. Execution Flow of ICM



(a) TimeWarp on messages M and vertex states S [10]



(b) ICM execution for 1 vertex in 1 superstep for above example

Figure 2. TimeWarp and ICM execution examples

end timepoint t_e (exclusive); and E is a set of *interval edges* $e = (eid, uid, vid, \tau_e)$, where *eid* is a unique ID for the edge, $uid, vid \in V$ are the source and sink vertices, and $\tau_e = [t'_s, t'_e)$ is the edge lifespan. The interval graph meets *uniqueness constraints* of its vertex and edge identifiers across time, and *referential integrity constraints* on the temporal existence of the source and sink vertices for any interval edge during its entire lifespan [10]. Figure 3(top) shows a sample interval graph, with vertices A–E and edges between them having weights for different intervals.

The *Interval-centric Computing Model (ICM)* [10] extends upon the Pregel *Vertex-centric Computing Model (VCM)* [25] to design time-dependent analytics over interval graphs. In VCM, the graph is partitioned and loaded into distributed memory of workers in a cluster. A user-defined `compute` logic executes in a data-parallel manner across vertices during an iteration (*superstep*); the logic updates each vertex's local state and may generate messages for its neighbouring vertices that are transferred in-memory or over the network at a synchronization barrier. Sink vertices receive these messages at the start of the next superstep, call `compute` on their prior state and the input messages, update their local state and generate messages. This iterates till convergence. Apache Giraph [1] is a popular open-source VCM platform.

ICM extends this concept to *interval vertices* that can update local states partitioned into intervals and generate interval messages [10]. It introduces a *TimeWarp* operator that temporally splits and aligns interval messages received for a vertex with its partitioned interval state, and executes `compute` for each resulting sub-interval (Figure 1). In Figure 2a, TimeWarp $⧖_{S\times M}$ on the vertex interval states

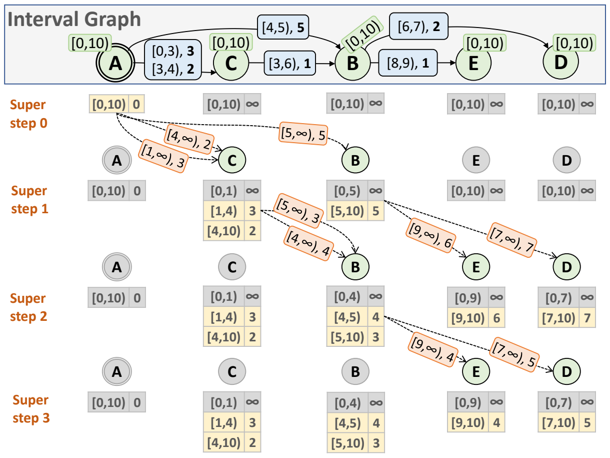


Figure 3. Temporal SSSP Execution using ICM [10]

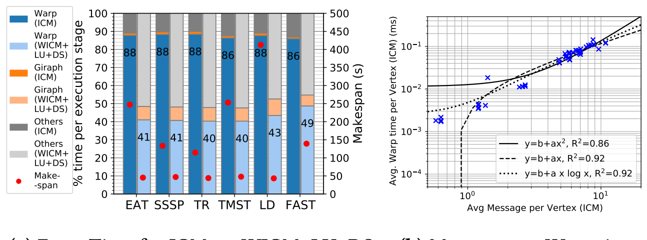
S and the incoming interval messages M is a set of triples with non-overlapping sub-intervals, having distinct vertex states and message sets that fall within the sub-interval: $\{\langle[0,2), s_1, \{\mu_1\}\rangle, \langle[2,3), s_1, \{\mu_1, \mu_2\}\rangle, \langle[3,4), s_2, \{\mu_1, \mu_2\}\rangle, \ldots\}$. The `compute` function is invoked on each of these (Figure 2b).

ICM also has a separate `scatter` user logic that triggers on out-edges of the vertex after each invocation of `compute`. It is preceded by a *TimeJoin* [36] between the updated interval vertex states and the interval out-edges to identify the edges whose lifespans overlap with the updated states' intervals. `scatter` generates interval messages for each edge based on the updated vertex state, and propagates it to the sink vertices at the barrier. The *TimeWarp*, `compute`, *TimeJoin* and `scatter` execute iteratively over multiple supersteps till convergence.

Figure 3 shows the execution of a temporal Single Source Shortest Path (SSSP) algorithm for a sample interval graph using ICM, with the associated interval vertex states at supersteps and interval messages passed between supersteps. The final vertex states converge to the shortest distances from the source for different time intervals in their lifespan.

3 Motivation and Approach

TimeWarp Overheads. TimeWarp is an operator introduced by ICM to help design temporal graph algorithms by making minimal changes to the user logic of their non-temporal VCM equivalent [10]. It also guarantees that the number of times the `compute` logic is called on an interval vertex is the theoretical minimum to avoid redundant computation. However, the overheads of TimeWarp itself mitigates these benefits. Figure 4a shows a stacked bar plot of the fraction of time spent in different execution phases in ICM (left bars) for six temporal traversal algorithms for the *Reddit social network graph* with 9.3 M vertices, 528.2 M edges and 122 timepoints (see Section 8 for details). The stacks highlight the



(a) Exec. Time for ICM vs. WICM+LU+DS **(b)** Messages vs. Warp time

Figure 4. Execution behavior for the Reddit Graph

major components of the overall runtime. TimeWarp (dark blue) is expensive and responsible for $\approx 86\%$ of the total execution time (or *makespan*). The `compute` and `scatter` user logic, message passing, superstep barrier, etc., collectively take only $\approx 10\%$ time (dark grey). Hence, optimizing the TimeWarp is critical.

TimeWarp has an initial *time-complexity* of $O(m \cdot \log(m))$ for temporally sorting the m messages received for a vertex. There is also an overhead for *replicating the incoming messages* to multiple sub-intervals for execution, which costs $O(m^2)$. Many ICM algorithms (Section 6.2) assign the interval of $[t_s, \infty)$ to messages generated by `scatter`, indicating that they are relevant to all the intervals of the sink vertex starting from t_s. Assuming that the m interval messages received by a vertex have a uniform distribution of their t_s within the lifespan of the vertex, warp would result in m sub-intervals. This causes the message with the smallest t_s to be replicated m times, the next smallest message $m-1$ times, etc. which gives us $(m + (m-1) + \ldots + 2 + 1) = O(m^2)$ message replicas. This replication causes memory and compute overheads within the JVM that is non-trivial, given the billions of interval messages processed for a large graph.

In Figure 2, the $m = 5$ input messages are warped with the 2 interval vertex states with a sort cost of $5 \cdot \log(5)$. The message $\mu_1 = 3$ is replicated for five intervals $[0,2), [2,4), [4,5), [5,9), [9,10)$, with a *replication factor* $\rho_{\mu 1} = 5$. Similarly, $\mu_2 = 4$ is replicated four times ($\rho_{\mu 2} = 4$), μ_4 thrice, μ_3 twice, and μ_5 has no extra replicas ($\rho_{\mu 5} = 1$). The *total message replicas* is $\rho^+ = \sum_{\mu \in M, \forall V}(\rho_\mu)$ and the *average replication factor* are $\overline{\rho} = \frac{\rho^+}{\sum_{M,\forall V}|M|}$, for the set of active vertices V in a superstep and the interval message set M per vertex. In Figure 2, $\rho^+ = 5+4+3+2+1 = 15 = O(m^2)$ and $\overline{\rho} = \frac{15}{5} = 3$.

Figure 4b shows a scatter plot between the average messages per vertex in each superstep (X-axis) and the average TimeWarp time per vertex (in milliseconds) for six graph algorithms from ten source vertices for Reddit, averaged over $\approx 21M$ invocations each. We also show three reference lines: $y = ax \cdot \log(x) + b$ [dotted], $y = ax^2 + b$ [solid] and $y = ax + b$ [dashed] with an R^2 correlation of 0.86–0.92, strongly indicating that TimeWarp depends on the message-count.

Scatter Overheads. Another design limitation in ICM is the eager execution of TimeJoin and `scatter` after *every*

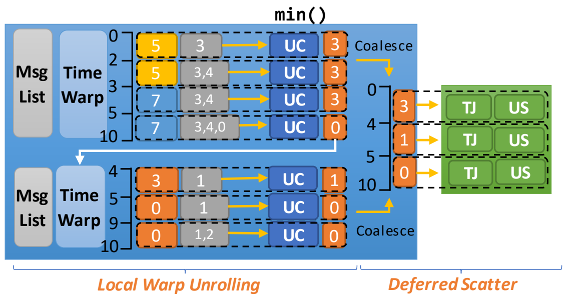


Figure 5. Execution of ICM with LU and DS Optimizations

`compute` call on a sub-interval. This allows pipelining of the delivery of the generated message to the sink vertex while the compute on the next sub-interval occurs. However, the output vertex states for adjacent sub-intervals may be identical. By executing `scatter` independently on each updated sub-interval immediately after a `compute`, we will have redundant calls to TimeJoin and `scatter` and also fragment the messages, with identical payload present in different (adjacent) intervals.

In Figure 2, for each of the 6 sub-intervals from TimeWarp, we execute `compute`, TimeJoin and `scatter` sequentially. Here, the output states from `compute` for the three adjacent intervals $[0, 2)$, $[2, 3)$, $[3, 4)$ have the same value of 3, and similarly, the sub-intervals $[5, 9)$, $[9, 10)$ have the same value of 0. Rather than performing TimeJoin and `scatter` six times and generating as many messages, coalescing adjacent sub-intervals having the same values into a single interval *after* all `compute` calls are done will help us reduce these to just 3 calls of *TimeJoin* and `scatter` for the coalesced sub-interval states $\langle[0, 4), 3\rangle$, $\langle[4, 5), 1\rangle$, $\langle[5, 10), 0\rangle$. These can also result in fewer messages.

Approach. *First, we leverage the intuition that reducing the average messages per-vertex will offer a significant reduction in the makespan for temporal graph algorithms designed using ICM.* We propose two complementary techniques to achieve this. One, we split the messages received at an interval vertex into smaller sets and invoke TimeWarp independently on each. We call this Local TimeWarp Unrolling (LU) (Section 4). Two, we temporally partition the interval graph and develop a windowed interval-centric execution model (WICM) to process the parts sequentially (Section 6). Both reduce the message count per-vertex, and hence ease the sort time-complexity and message replicas for TimeWarp.

Second, we defer the `scatter` execution (DS) and have it happen after *all* calls to `compute` for an interval vertex are finished (Section 5). Further, TimeJoin and `scatter` are called after the adjacent sub-intervals with identical output state values are coalesced. This reduces the calls to TimeJoin and `scatter`, and thus the number of interval messages.

E.g., in Figure 4a, the % execution time spent in TimeWarp using WICM+LU+DS (right bars, light orange) has reduced to ≈ 40% from ≈ 86% for ICM, and the makespan (red circles) has reduced by ≈ 68% compared to ICM. Our techniques also reduce the message count on average – a modest 6% drop from ≈ $551M$ to ≈ $518M$ for Reddit, but a larger ≈ 38% drop across all graphs.

4 Local TimeWarp Unrolling (LU)

Local TimeWarp Unrolling (LU) attempts to mitigate the super-linear sorting cost of the TimeWarp operator and the message replication costs, with the cardinality of messages received at a vertex. Instead of invoking warp on all the messages received at a "local" superstep in a single shot, we unroll the messages into smaller message sets and independently apply warp on each message set. This reduction in the size of the messages per warp will reduce the sorting time and, potentially, the number of message replicas as well.

In LU, we partition the input message list for a vertex into different sets and then invoke warp on each set and the vertex's state. Specifically, we take the outputs from TimeWarp on the first message set and the vertex state and invoke `compute` on each output tuple. Then, we take the updated vertex states after these `compute` calls and use them in the TimeWarp operation on the next message set, and so on. E.g., in Figure 5(left), we see TimeWarp invoked on the set $\mu_1 = 3, \mu_2 = 4, \mu_3 = 0$ with the states 5, 7 to get 4 output tuples, on each of which `compute` is called, which happens to be a *min* function on the messages and the state. These update the vertex states to 3, 0 for relevant sub-intervals based on the user logic. TimeWarp is then called on the set $\mu_4 = 1, \mu_5 = 2$ with the updated states 3, 0, and returns 3 output tuples on which `compute` is called.

There are different ways in which we might partition the messages into sets. Later in WICM, where a similar technique is applied at a coarser "global" granularity to split the interval graph into windows, we use more sophisticated approaches since its performance is much more sensitive to the splits. However, for LU, using an intuitive approach of splitting the message into k (approximately) equal-sized sets suffices. Since there are no message-ordering guarantees for VCM, we follow a FIFO order to form the sets. While the message sets expose additional data parallelism, there is already adequate parallelism present across vertices in a worker.

The time-complexity benefits from this technique are evident. Say a vertex receives m messages, and there are k sets. The size of each set will be ≈ $\frac{m}{k}$. The total sorting cost for the message sets becomes: $k \cdot (\frac{m}{k} \cdot \log \frac{m}{k}) = m \cdot \log m - m \cdot \log k$. If we set $k = \sqrt{m}$, this becomes $\frac{m \cdot \log m}{2}$, which is half of the sorting cost of the default ICM approach. As an optimization, if the number of messages m is smaller than some threshold m_{min}, we do not perform any warp unrolling, i.e., $k = 1$. In Section 7 we argue that such warp unrolling does not affect the correctness of the interval-centric graph algorithm.

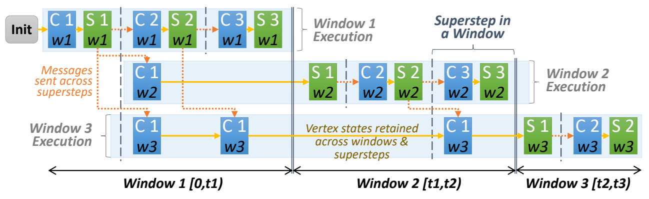


Figure 6. Execution Flow of Windowed ICM (WICM)

5 Deferred Message Scatter (DS)

ICM's eager execution model of executing `scatter` immediately on overlapping out-edges after each `compute` on a sub-interval can cause redundant processing and communication. In deferred message scatter (DS), we lazily evaluate TimeJoin and `scatter` after *all* calls to `compute` have completed for a vertex in the current superstep. In particular, the interval vertex state data structure is designed such that multiple sub-intervals that are temporally adjacent and have the same state value will coalesce. The new sub-interval will be their union and will be mapped to the shared value.

The updated execution flow within a superstep is shown in Figure 5(right). The output vertex states after all the `compute` calls are coalesced, and result in the interval states $\langle[0,4),3\rangle$, $\langle[4,5),1\rangle$, $\langle[5,10),0\rangle$. Specifically, the sub-intervals $[0,2)$, $[2,3)$, $[3,4)$ having state 3 have coalesced into $\langle[0,4),3\rangle$, and similarly, $[5,9)$, $[9,10)$ with value 0 have coalesced into $\langle[5,10),0\rangle$. Now TimeJoin results in 3 tuples (compared to 6 in Figure 2b) and just three calls to `scatter`. DS can be naturally coupled with LU, but can also operate independently if required. With LU, the call to `scatter` is deferred until all calls to `compute` from all the message sets are complete.

DS may seem counter-intuitive since we are replacing a pipelined execution of `compute`, `scatter` and message communication in ICM for different sub-intervals (Figure 2b) with one that introduces a local barrier between all `compute` calls and `scatter` for a vertex (Figure 5). However, this results in a significant reduction in the total quanta of work performed for that vertex. DS *guarantees* that the TimeJoin occurs only once for the vertex on its final states in that superstep, and that the number of `scatter` calls is the same or, often, fewer. Since each call to scatter usually generates one interval message, this also reduces the number of messages generated. Further, given that each worker has millions of vertices that can be executed in a data-parallel manner, there will continue to be collective pipelining of the compute/scatter threads that generate messages from different vertices, and the communications threads that deliver these messages to remote workers. In Section 7, we argue that DS offers results equivalent to ICM for a large class of temporal graph algorithms.

6 Windowed ICM (WICM)

The LU and DS techniques make local changes to a superstep execution of ICM. However, we can gain additional computation and messaging benefits by revising the execution model through a global unrolling of the interval graph's execution. *Windowed ICM (WICM)* is a complementary technique that divides the interval graph into windowed interval graphs. Each *windowed graph* has a disjoint sub-interval of the original graph. These windowed graphs are processed sequentially using ICM. Unrolling the execution across multiple sequentially-dependent windows reduces the message load per superstep in a window and the warp costs.

Such a redesign poses distinct challenges.

1. We must ensure that ICM and WICM give equivalent outputs for the temporal graph algorithms.
2. Like for LU and DS, there should be minimal/no changes to the user-defined `compute` and `scatter` logic for seamless reuse of existing ICM algorithms.
3. Not all window partitions will give the same performance benefits. We need sound heuristics to split the interval graph into the right-sized windows.

Section 7 addresses the first two points by formally modelling the execution behaviour of ICM, LU, DS and WICM, and providing proof of their equivalence. The constraints hold for a large class of temporal traversal algorithms, as illustrated in Section 6.2. We develop heuristics in Section 6.4 that use the graph topology to determine the ideal number of windows and their temporal boundaries.

6.1 WICM Execution Model

WICM operates on one windowed graph at a time. In each subgraph, WICM executes similar to ICM, performing iterative TimeWarp and `compute` on each interval vertex, followed by the TimeJoin and `scatter` on each out-edge. This is shown in Figure 6, where for Window 1 that spans from $[0, t_1)$, we have three supersteps of ICM execution of `compute` $(w1 - c1, c2, c3)$ and `scatter` $(w1 - s1, s2, s3)$.

However, interval messages generated by the `scatter` of one window may be destined for the interval vertices that appear in a future window. E.g., in Figure 6, the first scatter of Window 1 $(w1 - s1)$ generates interval messages that overlap with Windows 2 and 3. One option is to buffer such messages and apply them on the vertices when its window starts. This strictly separates the windows of execution. However, with billions of such messages, buffering can be costly.

We perform an *eager evaluation* on these messages by executing the `compute` for a future window as part of the current window's superstep, and update their local vertex states. We *do not* call the subsequent `scatter` on such eager compute execution. Instead, we mark those vertex states as having been updated and call `scatter` when the relevant window is scheduled for processing. The updated states are much more compact than buffering the messages. When it is the future window's turn to be processed, we resume ICM execution of the sub-graph from this updated vertex state. A vertex from a future window can have many

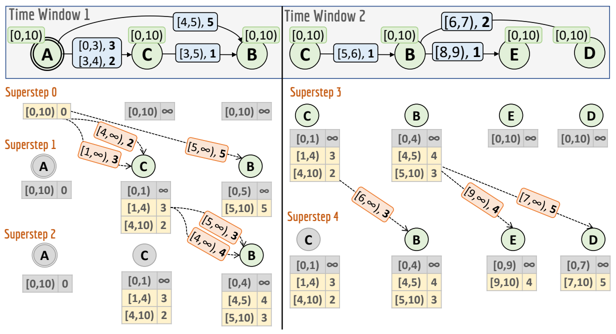


Figure 7. Temporal SSSP Execution on WICM

such eager compute executions from multiple supersteps of past windows.

This is illustrated in Figure 6. $w1 - s1$ generates interval messages for Window 2 and 3, which is processed in the next superstep by executing `compute`, $w2 - c1_1$ and $w3 - c1_1$, on vertices in those windows. Later, $w1 - s2$ again generates messages for Window 2, which is processed by $w3 - c1_2$. This repeats for $w2 - s2$, which generates messages for Window 3, executed by $w3 - c1_3$. When Window 3's execution starts, the updated vertex states from $w3 - c1_1, c1_2$ and $c1_3$ serve as a proxy for the execution of `compute` in the first superstep for that window, and we directly call the `scatter` for the first superstep, $w3 - s1$. If none of the vertices of a window has undergone eager evaluation, the starting superstep of that window's execution begins with `compute`.

Intuitively, spreading computation and message propagation across windows reduces the messages received per vertex, and offers a reduction in TimeWarp costs and message duplications. This is analyzed in Section 6.3. Further, WICM is complementary to LU and DS and can be combined. The benefits of using WICM and LU together may be limited since both aim to reduce the TimeWarp overheads.

6.2 Temporal Path Algorithms using WICM

Algorithm 1 offers a template of the `compute` and `scatter` logic for a broad class of temporal path algorithms designed using WICM. The traversal algorithms vary only in their user compute (⊕) and user scatter (σ) functions shown in blue. The algorithm is *identical* to ICM's, but the execution flow is different, as illustrated for *temporal SSSP*. In Figure 7, the interval graph from Figure 3 is split into two windows, $[0, 5)$ and $[5, 10)$. For Window 1, superstep 0 execution is similar to ICM Figure 3. In superstep 1, when vertex B receives an interval message from A, it updates its local state using `compute` but does not invoke `scatter` since both its out-edges fall outside the current window. This again happens in superstep 2 when B receives messages from C. Once execution on Window 1 has converged, we start execution of Window 2. Here, since the local states for B and C that overlap with Window 2 were updated, either through normal execution (state $\langle [4, 10), 2 \rangle$ for C) or eager execution

Algorithm 1 Temporal Path Traversal Algorithm Template

```
1: function COMPUTE(Vertex $v$, Time $t$, Messages[] $msgs$)
2:     vState ← $s_v(t)$, $b_v(t) \leftarrow 0$
3:     for Message $m$: $msgs$ do
4:         vState ← vState ⊕ $m$                ▷ User compute fn.
5:     end for
6:     if (vState ≠ $s_v(t)$) then
7:         $s_v(t)$ ← vState, $b_v(t) \leftarrow 1$
8:     end if
9: end function
1: function SCATTER(Edge $u \rightarrow v$, Vertex $u$, Time $t$)
2:     $t_r \leftarrow t + \delta(u \rightarrow v, t)$
3:         ▷ For time-backward algorithms, this becomes $t - \delta(u \rightarrow v, t)$
4:     $\mu \leftarrow \sigma(s_u(t), \pi(u \rightarrow v, t))$          ▷ User scatter fn.
5:     SENDMESSAGE($\mu_{u(t) \rightarrow v(t_r)}$)
6: end function
```

Table 1. User compute and scatter functions for algorithms

Algorithm	User Compute ⊕	User Scatter σ
EAT [43]	min()	$t + \delta(u \rightarrow v, t)$
SSSP [43]	min()	$s_u(t) + w(u \rightarrow v, t)$
TR [44]	max()	$s_u(t)$
FAST [43]	max()	$s_u(t)$
LD [43]	max()	$t - \delta(u \rightarrow v, t)$
TMST [17]	min()	(vid, $t + \delta(u \rightarrow v, t)$)

(state $\langle [5, 10), 3 \rangle$ for B), we directly trigger TimeJoin and `scatter` over their out-edges to initiate the propagation of their "updated" state. Compared with ICM, WICM results in fewer operations: 6 TimeWarp operations, 8 `compute` calls, and 8 `scatter` calls.

The equivalence proof for WICM with ICM in Section 7 places constraints on the user `compute` and `scatter` functions, ⊕ and σ. However, a large class of temporal traversal algorithms in literature can be designed within these limits. Table 1 lists the different functions that can be plugged into Algorithm 1 to design diverse temporal traversals. There is *no change* in the user logic between ICM and WICM for these algorithms [10], allowing for their rapid reuse.

Traversal algorithms like *Earliest Arrival Traversal (EAT)* [43] and *Temporal SSSP* [43] use `min` as the ⊕ function to select the shortest travel duration or distance. During σ, EAT propagates the arrival time for reaching the sink vertex from the source, while SSSP sends the cost for the new path by adding the edge weight. *Temporal Reachability (TR)* [44] and *Fastest Path (FAST)* [43], on the other hand, use `max` as the ⊕ operator. Their σ sends their current vertex state as a message to the sink; TR's message indicates that the sink is also reachable, while in FAST, σ passes the latest departure time from source, which is maintained as its vertex state.

One characteristic of these temporal algorithms is that they are time respecting in nature, i.e., the vertex state at a timepoint only influences other vertex states at future or earlier timepoints, but not both. *Latest Departure Time (LD)* [43] is a traversal algorithm that is time-respecting in the *backward* direction but otherwise similar to EAT.

Temporal Minimum Spanning Tree (TMST) [17] is similar to EAT with the variation that vertex state is a tuple, (vid, t), having the parent vertex ID and earliest arrival time from the source. $\oplus$ compares tuples by first comparing arrival times and then comparing the parent vertex ids if the arrival times are equal, to select the smallest. σ propagates the current source vertex ID and the arrival timepoint at the sink.

Besides these, there are also non-traversal algorithms like Temporal Triangle Count and Clustering Coefficient designed using an interval-centric model [10]. While $\oplus$ for both are commutative, associative and distributive, they are not selective functions. So these neighbourhood algorithms cannot reuse the same ICM user logic and require modifications for WICM. In particular, information on neighbouring vertices acquired during traversals needs to be remembered across windows. E.g., in temporal triangle count, vertices should retain the one-hop neighbourhood seen in past windows to test if they form time-respecting 3-cycles. We leave their design and evaluation as future work.

6.3 Reduced TimeWarp Overheads in WICM

WICM splits the temporal graph processing into w different windows. Splitting helps reduce both the warp computation and the message replication overheads. The `compute` operation in a window is pre-dominantly called only for messages that fall in that window, other than some eager evaluation of the first superstep `compute` for messages destined for future windows. So the number of incoming messages at a vertex will nominally decrease from m to $\frac{m}{w}$ when WICM splits a graph processing into w windows. Therefore, the *warp overheads per window* are given by the warp compute cost $\frac{m}{w} \cdot \log(\frac{m}{w})$ and the message replication cost $(\frac{m}{w})^2$. Since there are w windows, the *total warp compute costs* for WICM are $w \cdot (\frac{m}{w} \cdot \log(\frac{m}{w})) = m \cdot \log(m) - m \cdot \log(w)$ — a reduction by $m \cdot \log(w)$ over ICM, and the *total message replication* is $w \cdot (\frac{m}{w})^2 = \frac{m^2}{w}$ — a reduction by a factor of w over ICM.

The exact number of messages processed by ICM and WICM are not identical, even though the final outputs are equivalent. This is because of eager evaluation of `compute` for future windows that folds in changes from multiple prior windows but does not `scatter` them immediately, reducing the count of messages propagated in WICM. Also, since the same interval message can now be split across multiple windows, this can cause duplicate messages to be created rather than being processed in a single `compute` call. We empirically observe that the average messages per-vertex do substantially and consistently reduce in WICM.

One side-effect of windowing is that the number of supersteps for the entire execution run can increase. However, the length of a window does not have a bearing on the number of supersteps it will take. In Figure 7, ICM takes 4 supersteps to complete SSSP, whereas WICM requires 5 supersteps – 3 and 2 supersteps over two windows. This can increase the per-superstep Giraph overheads. However, as seen in Figure 4, the Giraph contribution is small compared to the benefits from the dominant TimeWarp stage, and our window partitioning heuristics avoid creating many small windows.

6.4 Heuristic for Partitioning into Windows

We use the analytical modelling to develop a heuristic to partition the temporal graph. This will select the *number of windows* w and the *time boundaries* for each window based on three key goals: (1) the messaging load in each window should be similar, (2) the number of messages that span windows should be reduced to mitigate redundant computation, and (3) the warp cost per window should be reduced.

For balancing the message load, we first calculate the *timepoint edge distribution* for the lifespan of the interval graph. This reports the number of interval edges that overlap with any timepoint. Since messages flow along the edges, we can balance the message load per window by ensuring that each window has a similar number of timepoint edges. This is in line with our goal #1 above.

We know the *TimeWarp overheads* are $m \cdot \log \frac{m}{w}$ using w windows in WICM and $\frac{m^2}{w}$ due to message replication. If the interval edges span across multiple windows, it causes redundant execution of the compute logic for those windows. We estimate this using the *edge overlap ratio*, defined as $\alpha = \frac{\sum_W \text{Interval edges in window } W}{\text{Interval edges in the graph}}$. This is a proxy for messages that span windows. Reducing this helps meet goal #2.

Our empirical analysis indicates that the impact of the message replication is much higher than warp operation. An edge overlap ratio of α causes the replication costs to nominally increase to $\frac{(\alpha \cdot m)^2}{w}$. Given m messages, the additional message replication cost is given by the metric $\beta = \frac{\alpha^2}{w}$. We select window boundaries that minimize β, addressing goal #3. In our experiments, α is in the range 1.09–1.97.

Algorithm 2 gives the heuristic for obtaining the time partitions for the windows. In FindWindows, we start with the timepoint edge distribution for the interval graph and its lifespan l. For every possible number of windows w, from 2 till $\sqrt{l}$, we evaluate all possible partitionings of the interval graph. This upper bound for w avoids too many fine-grained windows where the Giraph overheads may dominate.

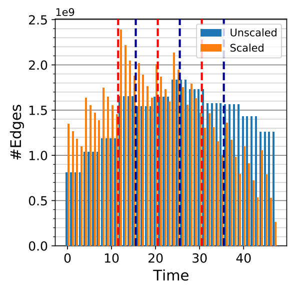
For a w, we select the window split boundaries using equi-depth histogram binning [28], i.e., the number of timepoint edges in each window is (nearly) the same. For this split, we evaluate the edge overlap ratio α and the metric β, and check if this split has a smaller β value than any previous split, and retain it if it is the best split seen so far.

E.g., Figure 8a shows the timepoint edge distribution for the WebUK graph, which has 133.6 M vertices, 5.5 B edges and lifespan of 48. Our heuristic chooses 4 windows with intervals $[0, 16)$, $[16, 26)$, $[26, 36)$, $[36, 48)$. The split boundaries are shown with a similar area under the curve for each window. The split gives $\alpha = 1.77$ and $\beta = 0.78$.

Algorithm 2 Window Partitioning Heuristic

```
 1: function EquiDepthHistogramBinning(Dist. $D$, $n$)
 2:     totalFreq ← Sum($D$)
 3:     appFreqPerBin ← $\frac{\texttt{totalFreq}}{n}$
 4:     $s \leftarrow [\,],\ f \leftarrow 0,\ b \leftarrow 1,\ i \leftarrow 0$
 5:     while $f <$ totalFreq do
 6:         if ($f > b\, *$ appFreqPerBin) then          ▷ Bin full
 7:             $s \leftarrow$ Add($s$, $i$), $b \leftarrow b + 1$
 8:         end if
 9:         $f \leftarrow f + D[i],\ i \leftarrow i + 1$
10:     end while
11:     return $s$
12: end function
```

```
 1: function FindWindows(Graph $G$)
 2:     $T \leftarrow$ GetTimepointEdgeDistribution($G$)
 3:     $\beta' \leftarrow \infty$
 4:     $l \leftarrow$ GetLifespan($G$)
 5:     for $w \in \{2 \cdots \sqrt{l}\}$ do
 6:         splits ← EquiDepthHistogramBinning($T$, $w$)
 7:         $\alpha \leftarrow$ GetEdgeOverlapRatio(splits,G)
 8:         $\beta \leftarrow \frac{\alpha^2}{w}$
 9:         if ($\beta < \beta'$) then          ▷ Found better split
10:             $\beta' \leftarrow \beta$, $w' \leftarrow w$, splits' = splits
11:         end if
12:     end for
13:     return $\langle w', splits' \rangle$
14: end function
```



(a) Distributions for WebUK

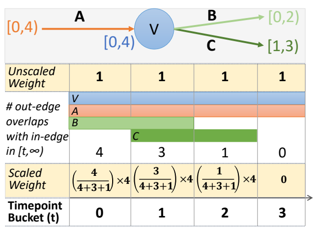

[0,4)
A
[0,4)
V
B
[0,2)
C
[1,3)
Unscaled Weight
1
1
1
1
# out-edge overlaps with in-edge in [t,∞)
V
A
B
C
4
3
1
0
Scaled Weight
$\left(\frac{4}{4+3+1}\right)\times 4$
$\left(\frac{3}{4+3+1}\right)\times 4$
$\left(\frac{1}{4+3+1}\right)\times 4$
0
Timepoint Bucket (t)
0
1
2
3
(b) Scaled Distribution Example

Figure 8. Timepoint Distribution

Finding the timepoint distribution takes a linear scan over all edges and binning them based on their lifespan, which has a time-complexity of $O(|E| + l)$ for $|E|$ interval edges in the graph and a graph lifespan of l. For Twitter, this takes just ≈ 3 *mins* using Apache Spark. The complexity of FindWindows is $O(l \cdot \sqrt{l})$. For the evaluated graphs, this takes < 200 *ms*. These can be run once, *a priori*, for an interval graph and reused across algorithms.

We introduce *two additional enhancements* to improve the quality of the partitioning strategy.

Scaled Distribution. First, we give different weights for the timepoints within an interval edge when estimating the timepoint distribution. Intuitively, since our temporal graph algorithms move forward in time, the propagation of a message arriving on an in-edge for a vertex depends on the number of out-edges whose lifespan are after the start time of the in-edge. We weight a timepoint in-edge based on the number of timepoint out-edges overlapping or following it, and normalize the weights such that the sum of the scaled weights for any interval edge remains the same as before. This *scaled timepoint edge distribution* function is returned by GetTimepointEdgeDistribution.

E.g., in Figure 8b, the vertex $V[0, 4)$ has one in-edge $A[0, 4)$, and two out-edges $B[0, 1)$, $C[1, 3)$. In the *unscaled timepoint distribution*, the in-edge A would contribute an equal weight of 1 to each timepoint bucket it overlaps with. With scaling, we see that at timepoint bucket 0, A overlaps/is followed by 4 timepoint out-edges, $B[0], B[1], C[1], C[2]$, giving it a weight of 4. At timepoint 1, A has a weight of 3 from out-edges $B[1], C[1], C[2]$; at timepoint 3, A has weight 1 from $C[2]$; and at timepoint 4, it has weight 0 as there are no out-edge overlaps. For each timepoint, we normalize and scale the weight by dividing it by the sum of all weights $(4 + 3 + 1 + 0)$ and multiplying it by the sum of unscaled weights for this in-edge $(1 + 1 + 1 + 1 = 4)$. So, A contributes a *scaled weight* of $\frac{4}{4+3+1+0} \times 4 = 2$ to the timepoint bucket 0, and a scaled weight of 0 to bucket 3. Figure 8a shows the scaled distribution (in orange) for WebUK. Using this, the heuristic still chooses 4 windows but with different spans: $[0, 12)$, $[12, 21)$, $[21, 31)$, $[31, 48)$. The time-complexity of calculating the scaled distribution is $O(l \cdot |E|)$, which is a one-time cost of ≈ 10–25 *mins* for our graphs using Spark.

Distribution Pruning. The second enhancement complements the static distribution derived above with runtime information on the *source vertex s* from which the temporal graph traversal will start. If the lifespan of the entire graph is $[0, l)$ and the lifespan of s is $[t_s, t_e)$, then instead of performing an equi-depth timepoint edge binning for $[0, l)$, we prune this to only the sub-interval of the graph starting from the existence of the source vertex, i.e., $[t_s, l)$. This avoids giving weightage to parts of the graph that will not participate in execution. This is suitably modified if the algorithm moves backwards in time, like LD. Likewise, we also bound maximum windows to $\sqrt{l - t_s}$. The re-binning for a given source vertex and recomputing the heuristic has a low overhead of < 200 *ms* for even our largest graphs, and can be applied at runtime. This applies to both the timepoint distributions.

7 Equivalence with ICM

ICM has two user-defined logic functions: `compute` and `scatter`. `compute` takes as input the partitioned messages and vertex states returned by TimeWarp and applies a user-defined operator $\oplus$ on these inputs. We assume that $\oplus$ is *commutative, associative* and *distributive*. Further, we also assume $\oplus$ to be a *selection* function i.e., it *selects* one of its input parameters as its output [41]; `scatter` takes as input the partitioned vertex state updated by `compute` and an edge over which messages will be sent. It sets the message payload as a function σ of vertex state and edge's properties. σ is assumed to be monotonic w.r.t the vertex state input.

7.1 Equivalence of LU+DS with ICM

Logically, ICM `compute` can be expressed as $\oplus(M \bowtie S)$ where S is the interval vertex-state and M is the list of messages.

Consider LU with 3 message sets. This can be expressed as $\oplus(M_1 ⧖ \oplus(M_2 ⧖ \oplus(M_3 ⧖ S)))$. Since $\oplus$ is distributive, associative and commutative, this can be reduced to $\oplus(M_1 ⧖ (M_2 ⧖ (M_3 ⧖ S)))$. For TimeWarp, we have $A ⧖ (B ⧖ C) = (A \cup B) ⧖ C = (A \cup C) ⧖ B = (C \cup B) ⧖ A$. As a result, we have $\oplus(\bigcup_i M_i ⧖ S) = \oplus(M ⧖ S)$, i.e., ICM and ICM+LU have an equivalent behavior.

Deferred scatter accumulates the output states from all calls to compute and coalesces adjacent intervals with the same value. Consider two such sub-intervals: $[t_1, t_2), \delta$ and $[t_2, t_3), \delta$ having the same state δ. Coalescing them will return $[t_1, t_3), \delta$. Since the message payload is a function of the state values, messages generated on these sub-intervals will have the same payload and will overlap: $[t_1, \infty), \mu$ and $[t_2, \infty), \mu$. In contrast, DS will generate only one message $[t_1, \infty), \mu$. Subsequently, TimeWarp will partitioned the former messages into: $[t_1, t_2), \{\mu\}$, and $[t_2, \infty), \{\mu, \mu\}$. However, the *selection* property of $\oplus$ resolves these partitions to $[t_1, t_2), \{\mu\}$, and $[t_2, \infty), \{\mu\}$. This is identical to applying TimeWarp and $\oplus$ on the single message, as in the latter case of DS.

7.2 Equivalence of WICM with ICM

The interval-centric computing can be reduced to a series of timepoint-centric computing, which operate on unit intervals. We formally argue the equivalence of *timepoint ICM (TCM)* and *timepoint WICM (WTCM)*. Since the key variations between TCM and ICM (WTCM and WICM) are the addition of TimeWarp, which does not vary the message payload, and execution interval-at-a-time rather than timepoint-at-a-time, the proof extends to the equivalence of ICM and WICM.

For a vertex u in the interval graph, $u(t)$ is the vertex at a *timepoint* t. The state value at any timepoint vertex $u(t)$ is denoted using $s_u(t)$. An edge between vertices u, v in the graph is defined as $u \longrightarrow v$. Edges can have *properties* at specific timepoints t, given as $\pi(u \longrightarrow v, t)$.

In the timepoint-based model, execution happens iteratively in supersteps composed of *ICM Compute* and *ICM Scatter*. In a superstep i, ICM Compute executes on every timepoint vertex $u(t')$ in the graph which receives one or more messages μ' and optionally updates its state from $s_u^{(i-1)}(t')$ to $s_u^i(t')$. ICM Scatter executes on every timepoint out-edge $(u \longrightarrow v, t)$ of a vertex u whose state was updated by ICM Compute, and generates a message μ to the sink vertex $v(t)$.

Let $\widehat{s}_v(t)$ be the final vertex state of $v(t)$ for TCM at the end of all supersteps. Let $\mathcal{U}_{R,v(t)}$ be the set of vertices which send any message to $v(t)$ over all supersteps. Using the commutative, associative and distributive property of the $\oplus$ operator:

$$\widehat{s}_v(t) = s_v^0(t) \oplus \left(\bigoplus_{u(t') \in \mathcal{U}_{R,v(t)}} \left(\bigoplus_{i \in \mathbb{I}_{u(t')}} \mu^i_{u(t') \rightarrow v(t)} \right) \right) \qquad (1)$$

where $s_v^0(t)$ is the initial vertex state at $i = 0$, and $\mathbb{I}_{u(t')}$ is the list of supersteps at which messages from $u(t')$ are received at $v(t)$. Since the temporal algorithms are forward moving, $t' < t$. Since $\oplus$ is a selection function and σ is monotonic, we can further reduce the equation as:

$$\widehat{s}_v(t) = s_v^0(t) \oplus \bigoplus_{u(t') \in \mathcal{U}_{R,v(t)}} \sigma(\widehat{s}_u(t'), \pi(u \rightarrow v, t')) \qquad (2)$$

Thus, the final state of any timepoint vertex $v(t)$ is a function of the final states of all timepoint vertices from which it receives any message. This holds both for TCM and WTCM. Therefore, TCM and WTCM are equivalent if their $\widehat{s}_v(t)$ and $\mathcal{U}_{R,v(t)}$ are equal for all timepoint vertices $v(t)$.

Let $\mathcal{U}_{S,v(t)}$ be the set of vertices to which $v(t)$ sends any message during execution. We use induction on t to show the equivalence of $\mathcal{U}_{R,v(t)}, \widehat{s}_v(t)$ and $\mathcal{U}_{S,v(t)}$ across TCM and WTCM. As base case, the initial vertex state for $t = 1$ is the same for vertices from TCM and WTCM; this state $v(1)$ never changes since it never receives a message as σ propagates messages forward in time. Assuming the hypothesis holds for all $t < T$, it is true for $t = T$ as follows:

- $u(t) \in \mathcal{U}_{R,v(T)} \Leftrightarrow v(T) \in \mathcal{U}_{S,u(t)}$. By induction hypothesis, $\forall v \forall t < T$, $\mathcal{U}_{S,u(t)}$ is equivalent for TCM and WTCM. Thus, $\mathcal{U}_{R,v(T)}$ is equivalent as well.
- If the state value is updated, scatter is called on all out-edges of the timepoint vertex for both TCM and WTCM. Thus, $\mathcal{U}_{S,u(t)}$ is equivalent.
- $\widehat{s}_v(t)$ is equivalent for TCM and WTCM using Eqn 2

8 Experiments

We run our experiments for diverse real and synthetic graphs, shown in Table 2. *Reddit* [14] is a social network where nodes are users and timestamped edges are the comments made on user posts. *Microsoft Academic Graph (MAG)* [35, 38] is a citation network where nodes are papers and edges represent citations. *WebUK* [5] is a union of web-graph snapshots from the .uk domain, where nodes are webpages and edges are links to other webpages. We duplicate and expand 12 monthly snapshots to 48 weekly ones. These are all real-world temporal graphs. *Twitter* aggregates 30 snapshots of the social network's topology in Sep. 2009 [6]. We add temporal variation through vertex/edge additions/deletions using the update distributions in Facebook's Link Bench tool [2]. The *LDBC graphs* [18] are synthetic ones from the Linked Data Benchmark Council (LDBC). We introduce temporal variations in these using the LDBC Datagen tool [42].

All of these are powerlaw graphs. They vary in the number of vertices (9.3 M to 113.6 M) and edges (528 M to 5.5 B). They also vary in their temporal properties, with a graph lifespan of 30–365 snapshots, an average vertex lifespan of 7–243 and an average edge lifespan of 8–152. For simplicity, all their edges have unit weight.

We evaluate the six temporal graph traversal algorithms described in Section 6.2: TR [44], SSSP [43], EAT [43], LD [43], FAST [43] and TMST [17]. In all of these, the algorithm starts from a source vertex. Since the performance is sensitive to the source, each algorithm is run from 10 random sources.

Table 2. Graph Characteristics

Graph	$\|V\|$	$\|E\|$	Avg. Vert. Lifespan	Avg. Edge Lifespan	Graph Lifespan
Reddit [14]	9.3M	528.2M	7.0	1.3	122
MAG [35]	67.4M	1.1B	17.8	12.9	219
WebUK [5]	133.6M	5.5B	16.3	12.5	48
Twitter [6]	52.5M	1.9B	27.5	8.1	30
LDBC-8_9-FB [18]	10.5M	848.6M	243.5	7.5	365
LDBC-9_0-FB [18]	12.8M	1.1B	69.8	44.7	104

LU and DS are implemented by modifying the TimeWarp and scatter phases of the Graphite Java ICM platform [10]. WICM is implemented as a light-weight wrapper over Graphite ICM. ICM and our optimizations are installed on 8 compute nodes of a commodity cluster with 1 additional node acting as coordinator. Each compute node has an Intel Xeon E5-2620 v4 CPU with 8 cores @ 2.10 *GHz* and 16 hyper-threads, 64 *GB* RAM and 1 *Gbps* network interface. They run CentOS 7 and the platforms use Java 8.

We compare our proposed techniques against the native Graphite ICM. ICM report consistently better performance [10] than the leading transformed graph model [43] and GoFFish [33] baselines. Hence, those are omitted from our comparisons. We evaluate four configurations:

- **ICM**: Native Graphite ICM
- **LU+DS**: Local warp unrolling and deferred message scatter optimizations included in ICM
- **WICM**: ICM with windowed execution model, and both scaling and pruning distribution based on source
- **WICM+LU+DS**: WICM with local warp unrolling and deferred message scatter optimizations

We limit our analysis to source vertices that take > 60 *secs* of runtime on ICM since benefits observed for our optimizations will be meaningful. For algorithm LD on MAG, all the sources finished within 60 *secs*. So, we omit LD on MAG in our results. In total, the experiments take $\approx 12k$ core-hours.

8.1 Benefits of LU+DS Optimizations for Native ICM

The *makespan reduction %* achieved by LU+DS, relative to native ICM, is given by $\frac{T(ICM)-T(LU+DS)}{T(ICM)} \times 100\%$. LU will reduce the messages per TimeWarp, and hence the time per warp and the overall contribution of warp's time to makespan. DS is expected to reduce the number of messages – this will reduce both the communication time and also the cost of warp on them. Our results confirm this. Next, we analyze the impact of LU/DS for the graphs and algorithms.

Figure 9(bars) show the average % reduction in makespan across all source vertices for the different techniques relative to ICM. The reduction for LU+DS compared to ICM is the yellow colored bar. The plots also report the average % drop in message count (red square) and the average % drop in warp time (blue triangle). The number of source vertices that take > 60 *secs* using ICM is a ♦ on the right Y axis. Values reported are averaged over these vertices. We also show a scatter plot of warp time reduction % against makespan reduction % in Figure 10a.

Reddit has a 44% average reduction in makespan for all six algorithms using LU+DS compared to ICM. This is exclusively due to LU – there is a 41% drop in time spent in TimeWarp but a negligible drop in message count due to DS. For *Twitter and WebUK*, we report a ≈ 60% drop in makespan for all algorithms but FAST. This is from a mix of message count reduction by ≈ 43%, which also mitigates warps, but we see an additional benefit from LU since the warp time reduction is higher at ≈ 63%.

For the *two LDBC graphs*, the makespan reduction is substantial at ≈ 89%, if we omit FAST. For both graphs, the benefits accrue due to a reduction in both messaging by ≈ 65% from DS and in warp time by ≈ 83% due to LU and DS. Interestingly, for the LD algorithm, LDBC-8, WebUK and Twitter show a much higher reduction in message count % compared to other algorithms, indicating the dominant benefits of DS.

MAG offers a mixed bag for different algorithms. While EAT and TMST are ≈ 54% faster due to fewer messages from DS, SSSP and TR offer no makespan benefits even though there is a ≈ 28% drop in message count and warp time compared to ICM. Interestingly, despite its larger size, algorithms run as fast on MAG as Reddit, taking an average of just 6 *mins* even using ICM.

The *FAST algorithm* is an outlier by offering diverse performance benefits. For Reddit and LDBC-9, FAST is quicker due to faster TimeWarps (both graphs) and fewer messages (LDBC-9). For Twitter and WebUK, we see no improvements since LU and DS are unable to reduce the warp time or the number of messages. The makespan for MAG and LDBC-9 is actually slower since the warp time increases using LU and DS, and there is no reduction in messaging. This is caused by the dueling effects of LU in reducing the per-warp time due to smaller message sets but increasing the number of warp calls due to multiple message sets. In the latter graphs, the net effect is negative. E.g., for LDBC-9, the time per warp reduces by 93% but we have a 15× increase in warp calls.

Large graphs have a substantial and meaningful reduction in wallclock time when using LU+DS. The average time per algorithm drops from 28 *mins* to 14 *mins* for Twitter; 52 *mins* to 37 *mins* for WebUK; 78 *mins* to 35 *mins* for LDBC-9; and a remarkable 59 *mins* to 6 *mins* for LDBC-8. We also achieve a modest drop in memory usage. For LU+DS, the average and peak memory used during execution is within 10% of ICM for all graphs except LDBC-8 where we see a ≈ 20% drop both in average and peak memory. This allows larger graphs to execute within the distributed memory of the cluster.

8.2 Performance of WICM

The orange bars of Figure 9 report the makespan reduction % of WICM relative to ICM, and its benefits are evident. It is *faster or comparable* to ICM for all graphs and algorithms,

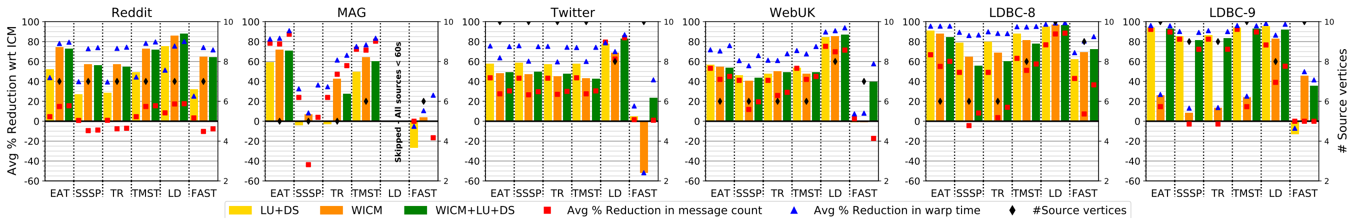
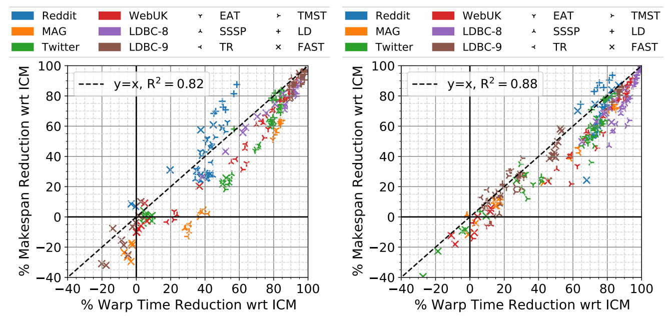


Figure 9. Bar plot of the *avg. Makespan reduction%* (left Y axis) of our techniques compared to ICM. The avg. % reduction in *message count* (🟥) and *warp time* (▲) are on the left Y axis. # of sources taking > 60*secs* using ICM (⧫) are on the right Y axis.



(a) LU+DS **(b)** WICM

Figure 10. Warp Time reduction% & Makespan reduction%

except for FAST on Twitter, and often gives an average reduction of 24 − 97% in makespan. For Twitter, WebUK, LDBC-8 and LDBC-9, with a runtime per algorithm of 2480 *secs* on average for ICM, we save 1360 *secs* per run on average.

As designed, the reduction in time taken by warp through windowed execution is the reason for this improvement in WICM. The makespan reduction % has a high correlation of $R^2 = 0.88$ with warp time reduction % in the scatter plot in Figure 10b. Excluding FAST on Twitter, all graphs and algorithms spend on average 62% less time on warp. This results both from the reduced $m \cdot log(m)$ analytical warp cost, which goes from 47.8*B* to 38.9*B* for WICM, and from fewer message replications, which fall from 76.2*B* to 31.1*B*.

For FAST on Twitter, we see an increase in the warp time for WICM compared to ICM. This is also accompanied by an increase in the numbers of messages due to spanning multiple windows. Unlike LU+DS where DS attempts to reduce the messages, WICM can increase the messages due to their duplication across windows. As a result, for SSSP and FAST on MAG, and for FAST on WebUK, there is minimal improvement for WICM over ICM since the reduction in warp time is offset by an increase in message count by 44–116%.

Combined WICM+LU+DS Technique. *When we combine WICM with LU+DS, it consistently outperforms ICM for all graphs and all algorithms.* This is seen in the green bars in Figure 9. Other than for a few cases, the combination of the techniques is comparable or better than each individual technique. It often tends to prefer the better of the WICM or LU+DS performance if they are divergent.

Specifically, WICM+LU+DS is able to address the outliers we observed for FAST with both WICM and LU+DS, where it gave poor performance for some graphs. There is no such case here and even in the worst case, we see MAG return a non-negative makespan reduction % for SSSP and FAST relative to ICM. The warp reduction comes both from WICM and LU, while the messages are reduced by DS. This resilience is the key value proposition for using the three techniques together, indicating their ability to generalize further to many other graphs and temporal traversal algorithms.

The memory usage here is similar to ICM, except for a 10% drop in memory for Reddit and 25% drop for LDBC-8.

8.3 Effectiveness of Window Partitioning Heuristic

We compare the makespan reduction % achieved by our heuristic that selects the number of windows and their partitioning boundary for WICM, against the *best makespan reduction %* achieved from sampling multiple window counts that span the lifespan of the graph. The sampled windows are partitioned using the scaled timepoint distribution but without the source vertex optimization. Specifically, for Reddit we sample and use window sizes of $\{6, 20, 30, 40\}$; for MAG $\{3, 5, 10, 20\}$; for WebUK $\{3, 6, 10, 20\}$; for Twitter $\{3, 5, 7, 10\}$; for LDBC-8 $\{10, 40, 70, 100\}$; and for LDBC-9 $\{3, 5, 15, 30\}$.

The makespan trends for the sampled window sizes (not shown) tend to follow a *U-curve*. As the number of windows increase, the warp time decreases and the Giraph overheads increase. But the benefits of warp outstrip the overheads. For large number of windows, the increase in the calls to warp out-weigh the reduction in time per warp. The makespan using our heuristic for WICM often falls near the trough (best window size) of the U-curve

Figure 11 shows the *average additional makespan reduction %* that is obtained when using the best WICM time from the sampled window sizes, compared to using our heuristic, relative to ICM, i.e., $\frac{T(WICM)-T(Best\ WICM)}{T(ICM)}$. The makespan reduction % is within 5% of the best for Reddit, WebUK and LDBC-8, and within 15% for MAG and Twitter, for all algorithms but FAST. This shows the robustness of our heuristic in practically exploiting the benefits of the optimizations.

For LDBC-9, our heuristic is under-performing. The best sampled window gives an extra 52% reduction in makespan for all algorithms except LD. For FAST, Reddit and LDBC-8 perform well but our heuristic is sub-optimal for the others.

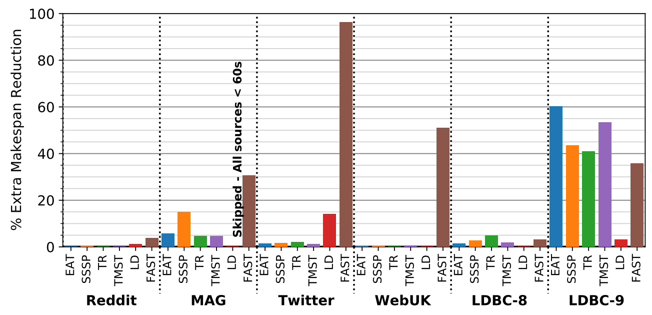


Figure 11. Under-achieved % Makespan reduction

9 Related Work

Distributed graph processing systems for static graphs grew popular with Google's Pregel [25], with its scalable vertex centric abstraction designed for Cloud and commodity clusters. Subsequently, other abstractions like edge-centric [31], subgraph-centric [33] and asynchronous models like Gather-Apply-Scatter (GAS) [24] were developed for static graphs. [27] summarising and comparing different vertex-centric frameworks. But these are not built to design time-dependent algorithms over temporal graphs.

Temporal graphs and its various representations have been discussed in literature [15, 16]. They can support both time-independent and time-dependent analytics. *Time-independent analytics* can operate snapshot at a time, and some systems [12, 21, 39] perform them efficiently by overlapping computation and messaging across individual snapshots. Some [39] automatically rewrite user-program using predefined rules to share computation and communication across snapshots. Chronos [12] optimizes the in-memory graph data layout and uses partitions for parallel processing. Others [21] store the changes between subsequent snapshots hierarchically to optimize snapshot retrieval and storage.

Time dependent analytics is more flexible as it allows the use of the graph's state across time, and can enforce temporal ordering within the algorithm. It is supported by a few platforms [10, 23, 32]. Tink [23] uses interval graphs and overlaps communication across snapshots but does not overlap computation. ICM's interval-centric abstraction removes redundant compute execution by performing temporal alignment within TimeWarp [10]. They also propose performance tuning like warp suppression and inline warp combiners. Our proposed optimizations are of a higher order and offer significant benefits. GoFFish [32] uses a snapshot representation of temporal graphs to perform time-dependent analytics. Graph states are shared with the next snapshot after processing the current snapshot. This has similarities with WICM where we eagerly update the future vertex states. However, WICM uses interval graphs and an interval-centric abstraction, with its programming simplicity and performance benefits.

Temporal path traversals have been discussed by Wu, et al. [43]. The authors propose efficient non-distributed algorithms for calculating the paths by unrolling the graphs across time, along with formal definitions of the paths and their complexity analysis. We focus on this class of traversal algorithms. Subgraph pattern matching on temporal graphs have also been proposed [22, 29]. Paranjape et al. [29] formalize the notion of motifs in temporal graphs and give a general approach to count any motif. Kumar, et al. [22] focus on counting temporal cycles using a two phased algorithm. Time dependent analytics in the form of path queries have also been explored [30, 44]. While some [44] discuss indexing mechanisms for answering whether certain types of time-respecting paths exist between two vertices in a given window, others [30] use ICM to answer path queries with temporal relations between adjacent nodes with no restriction on the path being time respecting.

There are frameworks for performing analytics on *streaming graphs* [4, 7, 8, 19, 26, 37, 41]. Raphtory [37] defines a streaming temporal graph model and proposes a graph management system for historical analysis along with update ingestion. Kickstarter [41] supports incremental computation on streaming graphs for a class of monotonic graph algorithms. The conditions for this class of algorithms is identical to the conditions required by WICM. Choudhury, et al. [8] explore subgraph pattern mining on streaming graphs using statistics collected from the streaming data.

GraphLab [24] uses eager evaluation in the form of asynchronous processing and immediately applying updates to the vertex. Maiter [47] on the other hand collects sufficient delta changes before applying them to the vertex states and propagating them forward. LWCP [45] proposes a lightweight checkpoint mechanism using vertex states and incremental updates for Pregel-like systems. All these are complementary to our optimizations.

10 Conclusions

We have examined the bottlenecks in the execution model of interval-centric computing, and proposed three optimization techniques to address the compute costs and message redundancies due to TimeWarp and eager evaluation of scatter. LU and DS operate within a superstep while WICM modifies the execution model. Our techniques rely on principled analytical modeling and further guarantee equivalence with ICM for the temporal graph algorithm outcome. Our detailed experiments over large realistic temporal graphs and algorithms highlight the significant and meaningful performance benefits from our techniques, applied separately and together, and correlate these improvements with the optimization goals.

As future work, we will extend this idea to operate on new windows of graph updates that arrive for a stream-based execution, and examine out-of-core execution using disk to cache the graph topology outside the current window.

References

[1] [n.d.]. Apache Giraph. https://giraph.apache.org/. [Online; accessed 9-October-2021].

[2] [n.d.]. Facebook Linkbench. https://github.com/facebookarchive/linkbench. [Online; accessed 21-July-2021].

[3] [n.d.]. TraceTogether. https://www.tracetogether.gov.sg/. [Online; accessed 21-July-2021].

[4] Laurent Bindschaedler, Jasmina Malicevic, Baptiste Lepers, Ashvin Goel, and Willy Zwaenepoel. 2021. Tesseract: distributed, general graph pattern mining on evolving graphs. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 458–473.

[5] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. 2008. A large time-aware web graph. In *ACM SIGIR Forum*, Vol. 42. ACM New York, NY, USA, 33–38.

[6] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna Gummadi. 2010. Measuring user influence in twitter: The million follower fallacy. In *Proceedings of the international AAAI conference on web and social media*, Vol. 4.

[7] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM european conference on Computer Systems*. 85–98.

[8] Sutanay Choudhury, Lawrence Holder, George Chin, Khushbu Agarwal, and John Feo. 2015. A selectivity based approach to continuous pattern detection in streaming graphs. *arXiv preprint arXiv:1503.00849* (2015).

[9] Kenneth L Cooke and Eric Halsey. 1966. The shortest route through a network with time-dependent internodal transit times. *Journal of mathematical analysis and applications* 14, 3 (1966), 493–498.

[10] Swapnil Gandhi and Yogesh Simmhan. 2020. An interval-centric model for distributed computing over temporal graphs. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1129–1140.

[11] Minyang Han and Khuzaima Daudjee. 2015. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proceedings of the VLDB Endowment* 8, 9 (2015), 950–961.

[12] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. 2014. Chronos: a graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*. 1–14.

[13] Bernhard Haslhofer, Roman Karl, and Erwin Filtz. 2016. O Bitcoin Where Art Thou? Insight into Large-Scale Transaction Graphs.. In *SEMANTiCS (Posters, Demos, SuCCESS)*.

[14] Jack Hessel, Chenhao Tan, and Lillian Lee. 2016. Science, askscience, and badscience: On the coexistence of highly related communities. In *Proceedings of the International AAAI Conference on Web and Social Media*, Vol. 10.

[15] Petter Holme. 2015. Modern temporal network theory: a colloquium. *The European Physical Journal B* 88, 9 (2015), 1–30.

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

[17] Silu Huang, Ada Wai-Chee Fu, and Ruifeng Liu. 2015. Minimum spanning trees in temporal graphs. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 419–430.

[18] Alexandru Iosup, Ahmed Musaafir, Alexandru Uta, Arnau Prat Pérez, Gábor Szárnyas, Hassan Chafi, Ilie Gabriel Tănase, Lifeng Nai, Michael Anderson, Mihai Capotă, et al. 2020. The LDBC Graphalytics Benchmark. *arXiv preprint arXiv:2011.15028* (2020).

[19] Xiaolin Jiang, Chengshuo Xu, Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2021. Tripoline: generalized incremental graph processing via graph triangle inequality. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 17–32.

[20] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. 2013. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European conference on computer systems*. 169–182.

[21] Udayan Khurana and Amol Deshpande. 2013. Efficient snapshot retrieval over historical graph data. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 997–1008.

[22] Rohit Kumar and Toon Calders. 2018. 2SCENT: an efficient algorithm for enumerating all simple temporal cycles. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1441–1453.

[23] Wouter Lightenberg, Yulong Pei, George Fletcher, and Mykola Pechenizkiy. 2018. Tink: A temporal graph analytics library for apache flink. In *Companion Proceedings of the The Web Conference 2018*. 71–72.

[24] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proceedings of the VLDB Endowment* 5, 8 (2012).

[25] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.

[26] Mugilan Mariappan and Keval Vora. 2019. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[27] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)* 48, 2 (2015), 1–39.

[28] Hamid Mousavi and Carlo Zaniolo. 2011. Fast and accurate computation of equi-depth histograms over data streams. In *Proceedings of the 14th International Conference on Extending Database Technology*. 69–80.

[29] Ashwin Paranjape, Austin R Benson, and Jure Leskovec. 2017. Motifs in temporal networks. In *Proceedings of the tenth ACM international conference on web search and data mining*. 601–610.

[30] Shriram Ramesh, Animesh Baranawal, and Yogesh Simmhan. 2020. A distributed path query engine for temporal property graphs. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 499–508.

[31] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 472–488.

[32] Yogesh Simmhan, Neel Choudhury, Charith Wickramaarachchi, Alok Kumbhare, Marc Frincu, Cauligi Raghavendra, and Viktor Prasanna. 2015. Distributed programming over time-series graphs. In *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 809–818.

[33] Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, and Viktor Prasanna. 2014. Goffish: A sub-graph centric framework for large-scale graph analytics. In *European Conference on Parallel Processing*. Springer, 451–462.

[34] Yogesh Simmhan, Tarun Rambha, Aakash Khochare, Shriram Ramesh, Animesh Baranawal, John Varghese George, Rahul Atul Bhope, Amrita Namtirtha, Amritha Sundararajan, Sharath Suresh Bhargav, et al. 2020. GoCoronaGo: privacy respecting contact tracing for COVID-19 management. *Journal of the Indian Institute of Science* (2020), 1–24.

[35] Arnab Sinha, Zhihong Shen, Yang Song, Hao Ma, Darrin Eide, Bo-June Hsu, and Kuansan Wang. 2015. An overview of microsoft academic service (mas) and applications. In *Proceedings of the 24th international conference on world wide web*. 243–246.

[36] Michael D Soo, Richard T Snodgrass, and Christian S Jensen. 1994. Efficient evaluation of the valid-time natural join. In *Proceedings of 1994 IEEE 10th International Conference on Data Engineering*. IEEE,

282–292.

[37] Benjamin Steer, Felix Cuadrado, and Richard Clegg. 2020. Raphtory: Streaming analysis of distributed temporal graphs. *Future Generation Computer Systems* 102 (2020), 453–464.

[38] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. 2008. Arnetminer: extraction and mining of academic social networks. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. 990–998.

[39] Manuel Then, Timo Kersten, Stephan Günnemann, Alfons Kemper, and Thomas Neumann. 2017. Automatic algorithm transformation for efficient multi-snapshot analytics on temporal graphs. *Proceedings of the VLDB Endowment* 10, 8 (2017), 877–888.

[40] Leslie G Valiant. 1989. *Bulk-synchronous parallel computers*. Harvard University, Center for Research in Computing Technology, Aiken ….

[41] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the twenty-second international conference on architectural support for programming languages and operating systems*. 237–251.

[42] Jack Waudby, Benjamin A Steer, Arnau Prat-Pérez, and Gábor Szárnyas. 2020. Supporting Dynamic Graphs and Temporal Entity Deletions in the LDBC Social Network Benchmark's Data Generator. In *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. 1–8.

[43] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. 2014. Path problems in temporal graphs. *Proceedings of the VLDB Endowment* 7, 9 (2014), 721–732.

[44] Huanhuan Wu, Yuzhen Huang, James Cheng, Jinfeng Li, and Yiping Ke. 2016. Reachability and time-based path queries in temporal graphs. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 145–156.

[45] Da Yan, James Cheng, Hongzhi Chen, Cheng Long, and Purushotham Bangalore. 2019. Lightweight Fault Tolerance in Pregel-Like Systems. In *Proceedings of the 48th International Conference on Parallel Processing*. 1–10.

[46] Da Yan, Yuzhen Huang, Miao Liu, Hongzhi Chen, James Cheng, Huanhuan Wu, and Chengcui Zhang. 2017. Graphd: Distributed vertex-centric graph processing beyond the memory limit. *IEEE Transactions on Parallel and Distributed Systems* 29, 1 (2017), 99–114.

[47] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. 2013. Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation. *IEEE Transactions on Parallel and Distributed Systems* 25, 8 (2013), 2091–2100.