Automatic Algorithm Transformation for Efficient Multi-Snapshot Analytics on Temporal Graphs

Manuel Then
then@in.tum.de

Timo Kersten
kersten@in.tum.de

Stephan Günnemann
guennemann@in.tum.de

Alfons Kemper
kemper@in.tum.de

Thomas Neumann
neumann@in.tum.de

Technical University of Munich

ABSTRACT

Analytical graph algorithms commonly compute metrics for a graph at one point in time. In practice it is often also of interest how metrics change over time, e.g., to find trends. For this purpose, algorithms must be executed for multiple graph snapshots.

We present *Single Algorithm Multiple Snapshots (SAMS)*, a novel approach to execute algorithms concurrently for multiple graph snapshots. SAMS automatically transforms graph algorithms to leverage similarities between the analyzed graph snapshots. The automatic transformation interleaves algorithm executions on multiple snapshots, synergistically shares their graph accesses and traversals, and optimizes the algorithm's data layout. Thus, SAMS can amortize the cost of random data accesses and improve memory bandwidth utilization—two main cost factors in graph analytics. We extensively evaluate SAMS using six well-known algorithms and multiple synthetic as well as real-world graph datasets. Our measurements show that in multi-snapshot analyses, SAMS offers runtime improvements of up to two orders of magnitude over traditional snapshot-at-a-time execution.

1. INTRODUCTION

Graphs are a natural representation for many real-world concepts like relationships in social networks, interactions in communities, and the topology of the Web. There has been growing interest in graph processing from both industry and academia which led to the creation of diverse analytical algorithms that extract knowledge from graphs. At the same time, various systems have been developed that efficiently execute predefined [18] as well as user-specified graph algorithms [5, 8, 15]. Most analytical algorithms operate on a static graph, i.e., the snapshot of a graph at a specific point in time. Consequently, existing analytical systems only optimize for this scenario.

Proceedings of the VLDB Endowment, Vol. 10, No. 8
Copyright 2017 VLDB Endowment 2150-8097/17/04.

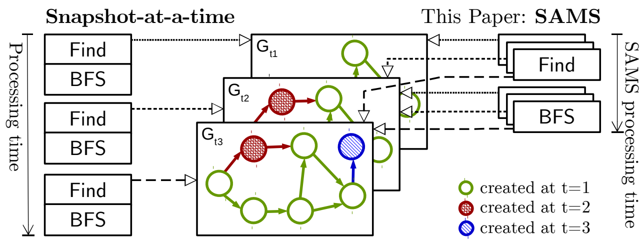


Figure 1: We propose Single Algorithm Multiple Snapshots (SAMS) to automatically interleave graph analytics algorithm executions on multiple snapshots such that they share graph accesses and traversals. SAMS significantly reduces the processing times for multi-snapshot analytics.

In many practical use cases, however, data analysts are interested in changes of graph metrics over time, e.g., to find trends and to project into the future. Consider, for example, analyzing the evolution of each user's importance in a social network. Using temporal graph information like vertex and edge creation times, a snapshot of the network can be reconstructed for each day of interest. Various existing ranking algorithms can then determine the users' importance in *each* of the snapshots. The rankings at multiple points in time can then be analyzed using standard data mining algorithms, e.g., to find the users who have been rising in importance most quickly, or to determine users that maintained a high rank over an extended period of time.

The execution of a graph algorithm for multiple snapshots is called multi-snapshot analytics, temporal graph mining [7], or snapshot-model stream analysis [23]; we use the first term to highlight that this work is not limited to temporal graphs or streams but applies to arbitrary graphs with snapshots. *Multi-snapshot analytics* adds the temporal dimension to numerous well-known, well-researched and readily-available graph analytics algorithms, that can not only determine importance metrics, but, for example, also find communities, weak links, or vertices and edges that fulfill specific criteria. Often, multi-snapshot analyses appear as part of complex analytical computations that comprise pre-computation steps, graph algorithm executions on snapshots that are based on the pre-computed data, and result aggregations which leverage data mining approaches like clustering or outlier detection.

By considering each snapshot as a separate graph and executing the algorithms on them independently, complex multi-snapshot analyses can be executed in existing graph analytics systems. This *snapshot-at-a-time* execution, however, leads to heavily redundant computation because the algorithms are executed repeatedly for very similar graphs—the snapshots of a *single* temporal graph. As a result, every execution exhibits the challenges that are inherent to graph algorithms, most notably their frequent random data accesses that lead to cache misses and compute stalls.

In this paper we present *Single Algorithm Multiple Snapshots* (*SAMS*), a novel automatic algorithm transformation that enables efficient execution of existing graph analytics algorithms in multi-snapshot graph analyses. SAMS reduces the runtimes of multi-snapshot analyses by exploiting structural similarities in snapshots of the same temporal graph. It automatically transforms graph algorithms that operate on a single graph snapshot such that they can be efficiently executed for multiple snapshots at the same time, letting the executions share common graph traversals and data accesses. Thus, SAMS greatly reduces data stalls and avoids redundant computation in multi-snapshot analytics.

Figure 1 visualizes the differences between snapshot-at-a-time execution and SAMS. It shows the scenario of executing a simple algorithm for three graph snapshots. For each snapshot, the algorithm searches a specific vertex using Find, and runs a breadth-first search, BFS, from it. Snapshot-at-a-time processing executes the algorithm separately for each snapshot and, thus, accesses graph elements multiple times that exist in more than one snapshot. In contrast, SAMS executes the algorithm for all snapshots at the same time, sharing graph and data accesses among multiple concurrent Find executions as well as among concurrent BFS executions. This allows SAMS to significantly reduce the overall processing time in multi-snapshot analyses.

To achieve concurrent algorithm execution and graph access sharing on multiple snapshots, SAMS automatically transforms graph algorithms in three steps: First, SAMS applies a rule-based transformation that interleaves multiple executions of the original algorithm such that they run concurrently and instruction-synchronously. We refer to these executions, that each operate on one graph snapshot, as algorithm instances. Second, SAMS replaces the instances' independent graph accesses with shared synchronized graph accesses. By means of synchronized graph access, all concurrently executed instances work on the same graph element, sharing their graph traversal logic, state, and data accesses. Third, SAMS adapts the algorithm's memory layout and introduces novel data structures that collocate the instances' concurrently processed data.

The fully-transformed SAMS algorithm efficiently computes the original algorithm's results on multiple graph snapshots. Exploiting structural similarities between the processed snapshots, SAMS greatly improves data locality, increases its cache hit rate, amortizes the cost of memory accesses, and significantly reduces CPU stalls. Furthermore, SAMS leverages redundancies in the executed program instances to improve code locality, avoid duplicate computation, and take advantage of modern CPUs' wide vector instructions. SAMS is the first approach that automatically optimizes existing graph algorithms for multi-snapshot analytics.

As the SAMS transformation does not change the instruction order within the concurrently executed instances, the original algorithm's conceptual and machine-specific optimizations are retained. Similarly, the original algorithm's degree of parallelism is unchanged because the concurrency of SAMS derives from our novel interleaving, not from the introduction of additional task parallelism.

While this paper focuses on in-memory graph analytics and its challenges, SAMS can also be applied to out-of-core and distributed processing. In out-of-core scenarios, SAMS can reduce the number of random disk accesses and decrease the amount of data that is accessed. For distributed graph analytics, SAMS facilitates merging data accesses that cannot be satisfied from the local data partition, and can, thus, reduce traffic and avoid high-latency network requests.

This paper's contributions are:

- We propose Single Algorithm Multiple Snapshots (SAMS) and describe how its synchronized concurrent execution allows shared data access and avoids redundant computations among program instances.
- We present formal transformation rules that automatically interleave multiple instances of a graph algorithm for instruction-synchronous concurrent execution. An implementation of our rule system is available for download.
- We describe how interleaved algorithms can automatically leverage structural similarities between snapshots through synchronized graph accesses and traversals.
- We describe how SAMS-optimized complex traversals and data structures can further improve the performance of multi-snapshot analyses.
- We extensively evaluate the performance of SAMS for widely-used graph analytics algorithms using multiple synthetic and real-world datasets. We show that SAMS exhibits up to two orders of magnitude speedup over snapshot-at-a-time execution.

The remainder of the paper is organized as follows. Section 2 introduces central terminology. Section 3 defines multi-snapshot analytics and compares SAMS with existing execution methods. Section 4 describes the automatic transformation of arbitrary graph algorithms to their SAMS variants. In Section 5 we evaluate SAMS. Section 6 gives an overview of related work, and Section 7 summarizes this paper.

2. BACKGROUND

This section introduces central terminology that is used in the rest of this paper.

2.1 Graph Analytics

We define a *graph* as a tuple $G = (V, E)$ of vertices $v \in V$ and edges $(s, d) \in E$ where $s, d \in V$ are the source and destination vertices of that edge, respectively. Vertices and edges may have named *properties*.

Analytical graph algorithms compute results that are based on the structure and properties of a graph. They commonly traverse the whole graph or significant parts of it. Examples of analytical graph algorithms include PageRank, triangle counting, and strongly connected components.

The main *challenges* in efficient graph analytics arise from the algorithms' inherent random data accesses. For large real-world graphs, these accesses often cannot be served from the fast but small CPU caches and, thus, cause high-latency main memory accesses that can stall the CPU and waste compute cycles. While random data accesses cannot be avoided in graph algorithms, their frequency can be reduced by improving the algorithms' spatial and temporal data locality [19].

2.2 Temporal Graph Analytics

Graphs evolve over time through the insertion and deletion of vertices and edges. We define *temporal graphs* as graphs which contain existence information for all vertices and edges at every point in time. Many real-world graphs are actually temporal because the creation times for vertices and edges are known. We refer to a temporal graph G at a point in time t as the *snapshot* G_t. Further, we define a *multi-snapshot graph* G^S as a graph in which vertex and edge existence information exists only for certain points in time. Multi-snapshot graphs are often induced by efficient updatable graph data structures [10, 14].

There has been much research about how the properties of graphs change over time, especially in the area of online communities and social networks. This work is especially based on the findings that graphs grow and densify over time [12, 13]. We leverage them by combining graph accesses to similar structures in multiple snapshots.

We distinguish two kinds of analytical algorithms that work on temporal graphs: temporal vs. time-agnostic. *Temporal algorithms* are aware of the graph's temporal nature and incorporate time-related information [17, 22]. In contrast, *time-agnostic algorithms* do not have a notion of time. They only yield sensible results when executed on a snapshot of a temporal graph. In this work we efficiently apply time-agnostic algorithms to multiple graph snapshots.

3. MULTI-SNAPSHOT ANALYTICS

In *multi-snapshot analytics*, a time-agnostic graph algorithm $\mathcal{F}$ is evaluated on multiple snapshots $S = \{G_{t_1}, \ldots, G_{t_n}\}$ of the same temporal or multi-snapshot graph G^S. The goal of multi-snapshot analytics is producing the results R_t for all snapshots S such that

$$\forall G_t \in S\colon\ R_t = \mathcal{F}(G_t)$$

We refer to the evaluation of the algorithm on a particular snapshot G_t as the *program instance* i_t. Conceptually, a program instance holds the algorithm's state and manages its execution on the instance's respective snapshot. By convention, we use the same subscript for program instances and the graph snapshots they process; for example, the algorithm execution for snapshot G_{t_1} is managed by program instance i_{t_1}. Furthermore, we define the set of *all program instances* $I = \{\ i_t \mid G_t \in S\ \}$.

3.1 Independent Snapshot Execution

Various execution strategies can be used to actually execute the program instances I. Two strategies are common in existing systems: First, the instances can be executed sequentially, *"snapshot at a time"*. Using this strategy, only one program instance is active at a time, so it can utilize all the system's resources to execute the algorithm for its respective snapshot. In particular, it can leverage the full CPU cache. Second, the instances can be executed *in parallel*, assigning each program instance a subset of the available CPU cores. As the program instances are independent, no inter-instance locking and communication overhead exists. However, when multiple independent program instances are active at the same time, they potentially thrash their shared caches, and hence may reduce their execution efficiency.

Both the snapshot-at-a-time and the parallel execution strategy have a common issue: They do not leverage that in multi-snapshot analytics the processed snapshots are created from the *same* graph at several points in time, and that they are, thus, likely *very similar*. Consequently, these execution strategies perform redundant data accesses and computations.

As an example, consider the execution of a simple 1-hop neighborhood traversal from v_0 on the two snapshots of G^S, depicted in Figure 2a, by the program instances i_{t_1} and i_{t_2}. G^S's solidly-outlined vertices v_0, v_1 and v_3 and their incident

Algorithm 1 1-hop neighborhood traversal.

```
1: $visit(v_0)$
2: for $n \in G^S.neighbors(v_0)$ do
3:     $visit(n)$
```

edges already exist at t_1, while the dashed vertex v_2 and edge b are added at t_2.

In Figure 2b we show snapshot-at-a-time execution, and visualize the order in which the program instances visit the graph's elements during the neighborhood traversal. The traversal is first executed completely for snapshot G_{t_1}, then snapshot G_{t_2} is traversed independently. Thus, the execution *lacks data access locality.* While this is not an issue for graphs as small as G^S, for real-world graphs with millions of vertices, and algorithms that access graph properties, this lack of locality can cause frequent cache misses which lead to CPU stalls. In addition, the figure shows that many of the vertices are accessed twice—once for each of the two processed snapshots. These *independent duplicate accesses* further exacerbate the execution's locality issues. Similar to snapshot-at-a-time execution, the parallel execution strategy also traverses the graph independently for the snapshots; hence, it also suffers from the locality and redundant access issues. As its data access pattern is similar to the former strategy, we omit it in Figure 2.

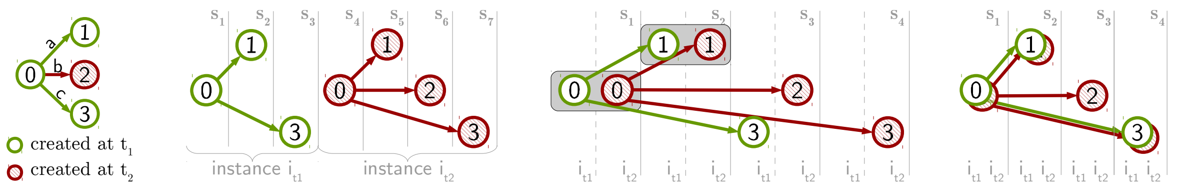
3.2 Single Algorithm Multiple Snapshots

Instead of processing them independently, we propose that program instances should be executed together concurrently and synchronized. In *synchronized concurrent execution*, multiple program instances are executed at the same time, so that they can synergistically exploit similarities in the processed snapshots and, thus, avoid redundant data accesses and computations. We propose *Single Algorithm Multiple Snapshots* (SAMS), an algorithm transformation that enables synchronized concurrent execution of an arbitrary graph algorithm's instances. The SAMS transformation comprises two major steps: *interleaving* the program instances' executions, and *synchronizing* their *graph accesses*.

Note that in this paper we use the term *concurrent* for statements that are processed on the same compute core in temporal proximity without any pre-determined order. SAMS does not introduce task parallelism; SAMS-transformed algorithms inherit the original algorithms' task parallelism.

3.2.1 Instance Interleaving

Interleaving transforms algorithms such that all of their instances concurrently execute the same algorithm statement. For a statement sequence $stmt_1; stmt_2$, this means that all interleaved instances concurrently execute $stmt_1$ before any instance processes $stmt_2$. Hence, interleaving can significantly improve code locality. Furthermore, when interleaved program instances concurrently access a graph element and



(a) Graph G^S (b) Snapshot-at-a-time (c) Interleaved (d) SAMS

Figure 2: Example multi-snapshot graph and execution sequence for Algorithm 1.

its properties, they can share this access, amortizing their memory access costs.

Consider again the example neighborhood traversal from v_0. Figure 2c depicts the order in which the two interleaved instances i_{t_1} and i_{t_2} of the traversal visit the vertices in G^S. The interleaved traversal starts at v_0, which is visited by both instances before its neighborhood is explored. Then, during neighborhood exploration, each instance iterates v_0's neighbor list in its respective graph snapshot:

$$i_{t_1} : G^S.neighbors(v_0) = [v_1, v_3]$$

$$i_{t_2} : G^S.neighbors(v_0) = [v_1, v_2, v_3]$$

v_1 is the first neighbor in both snapshots, so the interleaved instances visit it concurrently. The second neighbor element does, however, differ between the instances, so in the second iteration i_{t_1} visits v_3, and i_{t_2} visits v_2. In the last neighborhood iteration, i_{t_2} visits v_3, and i_{t_1} is idle as it already finished traversing its neighbor list.

Figure 2c highlight steps s_1 and s_2. In these steps shared graph element accesses are possible as a result of interleaving. We see that while interleaving is always beneficial for code locality, without further modifications it only allows access sharing for very similar graph structures. Thus, for SAMS we propose *graph-synchronous* interleaved execution.

3.2.2 Graph-Synchronous Interleaved Execution

Graph-synchronous interleaved execution synchronizes the interleaved program instances, such that all concurrently executed statements process the same graph element. Hence, it ensures maximum data access sharing between the instances.

As a processed graph element may not exist in all of the instances' respective snapshots, we introduce the concept of active instances. Intuitively, we call a program instance *active* for a given statement *stmt* when an equivalent single-snapshot execution of this instance would execute *stmt* in the same context. In Section 4.1 we give formal rules to determine when a program instance is active, and show that the concept of active instances is also important for the interleaving of complex control flows.

Algorithm 2 is the neighborhood traversal algorithm transformed to SAMS, i.e., interleaved and with synchronized graph accesses. Figure 2d shows the order in which its execution visits G^S's vertices. Because the instances i_{t_1} and i_{t_2} always process the same vertex, the SAMS algorithm also

Algorithm 2 SAMS variant of Algorithm 1 that allows its concurrent execution for two graph snapshots.

```
visit($v_0$, $\{i_{t_1}, i_{t_2}\}$)
for $(n, active_n) \in G^S.neighbors_*(v_0, \{i_{t_1}, i_{t_2}\})$ do
    visit($n$, $active_n$)
```

traverses the same neighbors list for them, using the set of instances $active_n$ to determine which are active for a neighbor n. SAMS further allows to merge the instances' *visit* calls, as is shown in the listing and the figure. In addition to the vertices v_0 and v_1 that were already visited concurrently by interleaved execution, SAMS is also able to correctly share the visit of v_3.

4. AUTOMATIC TRANSFORMATION TO SAMS EXECUTION

Time-agnostic graph algorithms analyze the structure and properties of a graph snapshot. They can be transformed to Single Algorithm Multiple Snapshots (SAMS) algorithms that concurrently compute the original algorithms' results for multiple snapshots and share common graph accesses. While this transformation can be done manually, SAMS is the first work that automatically transforms algorithms for efficient multi-snapshots analytics, thus, also allowing its use in ad-hoc queries.

In this section we describe the three steps of the SAMS transformation. First, we explain formal transformation rules to *interleave* graph algorithms such that multiple of their instances can be processed concurrently and instruction-synchronously, improving the algorithm's code locality as well as its data locality for scalars. Second, we elaborate the *synchronization* of the instances' *graph accesses* to improve the algorithms' temporal data locality for graph structure traversals and property accesses. Finally, we propose a SAMS-optimized *memory layout* and specialized *data structures* that greatly improve the the algorithms' spatial locality and cache utilization for multi-snapshot analytics.

4.1 Interleaving of Program Instances

For algorithms that neither contain branches nor use local variables, interleaving can simply be done by duplicating each algorithm's statements for every instance. However, once non-trivial control flow and variables are considered, this does not work because local variables can have different values in the program instances, and because branch predicates may evaluate differently, leading to differing runtime control flows in the instances. Furthermore, duplicating branches and loops does not interleave their contents.

We propose statement-specific transformation rules, shown in Table 1 that overcome these issues by managing *local state per program instance*, determining for each statement which *program instances* are *active*, and translating *control flow to data flow*. As each rule is responsible for interleaving one type of statement, Table 1 also implicitly defines the constructs of the language used in this paper. The statements' semantics closely follows that of commonly-used imperative languages, e.g. the IMP language described in [21], augmented with graph-specific collections and iterations over collections that are similar to the GreenMarl language [8].

We implemented all interleaving rules using the rewriting logic tool Maude [3]. Our implementation is available online[1] and includes a graph algorithms syntax that resembles the one used in this paper, executable semantics for this syntax, and our rules to automatically perform the interleaving. Furthermore, it contains the example discussed in this section and other graph algorithms.

We use Algorithm 3 as the running example in this section. Similar to a single PageRank iteration, the *SumNeighbors* function sums the global property P for each vertex's neighbors into a temporary property R, which it returns.

Algorithm 3 Property aggregation algorithm.

```
1: fun $SumNeighbors(G)$
2:     for $v \in G.vertices()$ do
3:         $R[v] := 0$
4:         for $n \in G.neighbors(v)$ do
5:             $R[v] := R[v] + P[n]$
6:     return R
```

The SAMS algorithm interleaving is based on the *interleaving marker* **sim** which tags statements that must be interleaved. By applying our transformation rules, statements that are marked for interleaving are successively transformed either to statements that are interleaved, or into partly interleaved statements which still contain interleaving markers and must, thus, be further transformed.

We start a graph algorithm's transformation by marking all of its root elements with **sim**. For most graph algorithms there is only a single root element: the actual algorithm function definition. Adding the interleaving marker to Algorithm 3 gives:

```
sim $\binom{-}{-}$ do
    fun $SumNeighbors(G)$
        [ algorithm implementation ]
```

This matches the left hand side of Rule 1 in Table 1. Applying this rule results in an interleaved function header and a body that is marked for interleaving:

```
fun $SumNeighbors_*(S, G_*)$
    $\mathcal{A}(SumNeighbors_*) := S$
    sim $\binom{S}{\mathcal{A}(SumNeighbors_*)}$ do
        [ algorithm implementation ]
```

In the remainder of this section we explain the transformation rules for all statements that may appear in function bodies. After the interleaving, $SumNeighbors_*$ may be executed for a set of program instances $A \subseteq I$, and guarantees the same results as snapshot-at-a-time execution:

$$R_* := SumNeighbors_*(A, G_*)$$
$$\Leftrightarrow\ \forall i_t \in A : R_{i_t} := SumNeighbors_{i_t}(G_t)$$

By convention, we use the asterisk $*$ on functions to denote that they are interleaved and can be executed for multiple program instances, and on variables to show that they hold values for multiple program instances. Thus, in this case, R_* contains values for all instances in I. In it, R_{i_t} is the variable R for program instance i_t. We further use subscript indices to refer to function evaluations or variable accesses in a specific program instance.

[1]http://db.in.tum.de/~then/data/SAMS-rules.tar.gz

Table 1: Rules for algorithm interleaving.

Rule 1 Function definition	**sim** $\binom{-}{-}$ **do**
	  fun $F(args...)$
	    stmt
		fun $F_*(S, args_*...)$
		  $\mathcal{A}(F_*) := S$
		  sim $\binom{S}{\mathcal{A}(F_*)}$ **do**
		    stmt

Rule 2 Statement sequence	**sim** $\binom{S}{D}$ **do** $stmt_1$ $stmt_2$	**sim** $\binom{S}{D}$ **do** $stmt_1$ **sim** $\binom{S}{D}$ **do** $stmt_2$
Rule 3 Expression evaluation and assignment	**sim** $\binom{S}{D}$ **do** $v := E$	$v_* := eval_*(E, S \cap D)$
Rule 4 Conditional branch statement	**sim** $\binom{S}{D}$ **do** **if** E **then** $stmt_t$ **else** $stmt_f$	$e_* := eval_*(E, S \cap D)$ $T := \{i \mid e_i = true\}$ **sim** $\binom{T}{D}$ **do** $stmt_t$ **sim** $\binom{S \setminus T}{D}$ **do** $stmt_f$

Rule 5 While loop	**sim** $\binom{S}{D}$ **do**
	  while E **do**
	    stmt
		$\mathcal{A}(\phi) := S$
		ϕ: **while** $D \cap \mathcal{A}(\phi) \neq \varnothing$ **do**
		  $e_* := eval_*(E, \mathcal{A}(\phi) \cap D)$
		  $T := \{i \mid e_i = true\}$
		  $\mathcal{A}(\phi) := \mathcal{A}(\phi) \cap T$
		  sim $\binom{T}{D \cap \mathcal{A}(\phi)}$ **do**
		    stmt

Rule 6 Loop ϕ break statement	**sim** $\binom{S}{D}$ **do** **break**	$\mathcal{A}(\phi) := \mathcal{A}(\phi) \setminus (S \cap D)$

Rule 7 For loop	**sim** $\binom{S}{D}$ **do**
	  for $v \in E$ **do**
	    stmt
		$\mathcal{A}(\phi) := S$
		ϕ: **for** $v_* \in collect_*(E, S \cap D)$ **do**
		  sim $\binom{S \cap \mathcal{A}(v_*)}{D \cap \mathcal{A}(\phi)}$ **do**
		    stmt

Rule 8 Function call	**sim** $\binom{S}{D}$ **do** $v := F(x...)$	$v_* := F_*(S \cap D, x_*...)$
Rule 9 Function ψ return statement	**sim** $\binom{S}{D}$ **do** **return** E	**sim** $\binom{S}{D}$ **do** $\mathcal{R}(\psi) := E$ $\mathcal{A}(\psi) := \mathcal{A}(\psi) \setminus (S \cap D)$ **if** $\mathcal{A}(\psi) = \emptyset$ **then return** $\mathcal{R}(\psi)$

We previously introduced the notion of active program instances for a statement as the instances that must execute it. The active program instances—or *active set*—is determined by the statement's scope as well as by interactions with other statements. For example, a **break** statement influences the active set of its respective loop. We use the interleaving marker **sim** to model the scope and inter-statement influences as scope constraints S and dynamic constraints D, respectively. Their details are discussed in Sections 4.1.2 and 4.1.3; there, we also introduce the active set access function $\mathcal{A}$. Marking a statement with **sim**

$$\textbf{sim}\ \binom{S}{D}\ \textbf{do}\ stmt$$

means that *stmt* must be executed for the program instances $A = S \cap D$. All instances must posess an exclusive program state so that executions are independant. Note however, that the interleaving marker is not meant to have *execution semantics*; thus, the algorithm is not executable in our framework as long as it contains interleaving markers.

4.1.1 Variables

In an interleaved algorithm, a variable v may have different values for multiple concurrently processed program instances. Thus, unless it can be inferred that all instances of v always have the same value, it is necessary to store an instance of v for each program instance.

Rule 3 uses the concurrent *expression evaluation function* $eval_*(E)$ to transform assignments such that multiple instances of the assigned variable are introduced. It evaluates the expression E for every active program instance $i \in A$:

$$v_* := eval_*(E, A) \;\Leftrightarrow\; \forall i \in A : v_i := eval_i(E)$$

Depending on the variable's data type and the evaluated expression, concrete implementations of $eval_*$ may use highly-tuned vectorized evaluation or fallback to evaluating the expression separately for each program instance. For program instances that are not active, the value of v_* is not changed.

Consider again Algorithm 3. Applying the assignment transformation to Line 5 yields:

$$R_*[v_*] := eval_*(R[v] + P[n], A)$$

Note how R_* denotes that one instance of the variable R is created per program instance, and how $eval_*$ operates on the original expression $R[v]+P[n]$.

4.1.2 Branches

Conditional branches are created using the **if** statement shown in Rule 4. It evaluates its predicate E and executes either $stmt_1$ or $stmt_2$, but never both.

In an interleaved algorithm the predicate may, however, evaluate differently for the active program instances, so that a subset $T \subseteq S$ must execute the *true* branch $stmt_t$, and all other instances execute the *false* branch $stmt_f$. Rule 4 models this by adding the scope constraints T and $S \backslash T$ to the respective branches:

```
sim $\binom{T}{D}$ do
    $stmt_t$
sim $\binom{S\backslash T}{D}$ do
    $stmt_f$
```

Scope constraints—the upper part of the **sim** marker—restrict the active sets of all statements within a *scope*, e.g., a specific branch or a loop. They are evaluated when the scope is entered and their resulting active set does not change within the scope. For optimizations it is important to note that scope constraints are monotonous: given a scope a that contains scope b implies for their active sets that $A_b \subseteq A_a$.

By means of this branch transformation, we transform control flow to data flow [1]. After the branch, the instances' potentially diverged control flows reconverge.

4.1.3 While Loops

while loop interleaving must solve two issues: First, loop conditions may evaluate differently in the active program instances. Second, loop bodies may contain **break** statements which immediately stop the respective instances' loop executions. Taking these issues into consideration, an interleaved **while** loop iterates as long as at least one program instance is still active for it.

Rule 5 shows our **while** loop interleaving. We handle loop conditions similar to conditional branches. The instance set T is computed before each iteration and used as the loop body's scope constraint. Thus, each instance is only executed as long as E evaluates to *true* for it. If T is empty the loop is exited.

In contrast, **break** statements may influence whether statements outside of their scope are active or not. Thus, they cannot be implemented by means of scope constraints. Instead, we create a new active set $\mathcal{A}(\phi)$ that tracks which program instances have not yet finished the **while** loop, and add it as a dynamic constraint to the loop body. *Dynamic constraints*, which are denoted in the lower part of the **sim** marker, are more general than scope constraints. They are allowed to influence the active sets of all subsequent statements and may change within a scope. This means, however, that dynamic constraints must—conceptually—be re-evaluated for every statement in the loop.

Rule 6 describes how **break** statements influence the active set of their respective loop using the *active set access function* $\mathcal{A}$. In nested loops, we distinguish the different active sets by assigning each loop a unique label ϕ. For loops that do not contain **break** statements, no dynamic constraint needs to be added, as we show in Section 4.2, Algorithm 4.

4.1.4 For Loops

Interleaved **for** loops concurrently iterate over the elements of a collection. As the interleaved neighbor iteration in Figure 2c visualizes, this collection may contain different elements for the active program instances. Hence, Rule 7 introduces the loop variable v_* which holds a value for each instance. For notational convenience, we introduce the function $collect_*$ which assigns values from the collection that each active program instance iterates, such that:

$$\forall a \in A : \quad x \in_a collect_*(E, A) \;\Leftrightarrow\; x \in eval_a(E)$$

We write $x \in_a C_*$ to denote that x is an element of C in instance a. To handle differing collection sizes between the active program instances, we further introduce the active set $\mathcal{A}(v_*)$ for the loop variable. In each loop iteration, $collect_*$ sets $\mathcal{A}(v_*)$ to all program instances for which a value was assigned to the loop variable. Specifically, program instances that are finished iterating their collections are not in the loop variable's active set. We add $\mathcal{A}(v_*)$ as a scope constraint on the loop's body to ensure that it is only executed by program instances that did assign a loop variable value in the current iteration.

As we described for the **while** loop, we add the dynamic constraint $\mathcal{A}(\phi)$ to handle **break** statements.

4.1.5 Function Calls

Rule 8 transforms function calls into calls to the same function's interleaved variant. In case it is not possible to build an interleaved variant of the function, we call the original function for every active program instance using $eval_*$. For example, this is necessary when a system function is called for which the source code is not available.

4.1.6 Return Statements

The return values of an interleaved function F_* are stored in the variable $\mathcal{R}(F_*)$, see Rule 9. It is set for program instances that execute a **return** statement, in which case that instance is also removed from the function's active set $\mathcal{A}(\psi)$ (where ψ is F_*'s identifier). Once the function's active set is empty, the function completed its execution.

4.1.7 Confluence and Termination

Algorithm transformations must be confluent, i.e., have the same result independently of the rule application order, and terminate, i.e., finish after a finite number of rule applications. The SAMS interleaving transformation presented in this section fulfills these requirements.

Intuitively, the rules are *confluent* because first, for every statement there is only one rule that can match it, and second, whenever there is a choice of which rule to apply, rule applications do not prevent any of the other choices from being applied later. Furthermore, the rule application must *terminate* because every rule application strictly reduces the number of language constructs that are left for transformation. Considering that all algorithms are finite by definition, successive rule application on any algorithm must reach a point at which no statements are left for transformation, which means that the transformation is finished.

4.1.8 Correctness

To demonstrate the correctness of the transformation rules we need to prove for every rule, that the program on the left-hand side (LHS) of the rule always produces the same result as the right-hand side (RHS). For brevity, we only show a proof sketch for Rule 7, as it covers all important aspects of SAMS transformations:

```
sim $\binom{S}{D}$ do                     $\mathcal{A}(\phi) := S$
    for $v \in E$ do           $\rangle$   $\phi$: for $v_* \in collect_*(E, S \cap D)$ do
        stmt                            sim $\binom{S \cap \mathcal{A}(v_*)}{D \cap \mathcal{A}(\phi)}$ do
                                            stmt
```

We need to show, that for every program instance i_t in which the LHS transforms the initial program instance state S_t into the final state S'_t, the RHS does so as well. The LHS contains the **sim** $\binom{S}{D}$ marker with a **for** loop. It indicates that the loop must be executed for every program instance $i_t \in S \cap D$. Let the result of evaluating the expression E in i_t be a list L containing the elements e_1 through e_n: $L = eval(E, i_t) = [e_1, ..., e_n]$. The **for** loop iterates over L and invokes the loop body *stmt* with the loop variable v bound to an element $e \in L$: $stmt[v = e]$. During loop execution, every invocation of *stmt* produces an intermediate state, leading to the final state S'_t:

$$S_t \xrightarrow{stmt[v=e_1]} S_{t,1} \;\ldots\; S_{t,n-1} \xrightarrow{stmt[v=e_n]} S_{t,n};\; S_{t,n} = S'_t$$

The RHS contains a **for** loop with a **sim** marker which contains *stmt*. We show that for every program instance the same sequence of state transitions is produced. Therefore, per Rule 7 let

$$S \cap D = \{i_1, ..., i_k\}$$
$$L_* = collect_*(E, S \cap D) = [(e_{1,1}, ..., e_{1,k}), ..., (e_{n,1}, .., e_{n,k})]$$
$$\mathcal{A} : L_* \to \mathcal{P}(S \cap D)$$

where $\mathcal{P}$ is the powersetfunction and $\mathcal{A}$ is a function from L_* to $\mathcal{P}(S \cap D)$ such that (as per definiton of *collect*$_*$)

$$I) \quad \forall i_t \in S \cap D.\; eval(E, i_t) = [\; l_{i_t} \mid l \in L_* \wedge i_t \in \mathcal{A}(l) \;]$$

the elements in L_* contain the values that would have been produced by *eval* for the program instances $S \cap D$, so that with respect to any specific program instance they are in the same order as produced by *eval*. As the number of results produced by *eval* may differ in the program instances, some values e_j may be invalid. An element $l \in L_*$ is only guaranteed to be valid for the active program instances $\mathcal{A}(l)$.

The RHS **for** loop ϕ executes its body

```
sim $\binom{S \cap \mathcal{A}(v_*)}{D \cap \mathcal{A}(\phi)}$ do
    stmt
```

for every $v_* \in L_*$. In each execution **sim**, according to its definition, changes the state of every program instance i_t

$$II) \quad i_t \in S \cap \mathcal{A}(v_*) \cap D \cap \mathcal{A}(\phi)$$

by invoking *stmt* on the program instance state $S_{t,j}$ to produce state $S_{t,j+1}$. In every invocation of *stmt*, v is bound to the element of v_* that corresponds to i_t: $v_{i_t} = e_{j,t}$:

$$S_{t,j} \xrightarrow{stmt[v=v_{i_t}]} S_{t,\, j+1}.$$

Due to I), in every program instance the same values as on the LHS are applied in the same order. Due to the restriction to $\mathcal{A}(v_*)$ in II), only valid values are assigned to v. Consequently, the RHS produces the same chain of transitions as the LHS within every program instance. □

4.2 Graph-Synchronous Execution

The second component of SAMS is graph-synchronous execution. *Graph-synchronous execution* leverages the structure of interleaved algorithms and ensures that all active program instances process the same graph element. It greatly reduces redundant memory accesses and computations, and, thus, further improves the efficiency of multi-snapshot algorithms.

Similar to existing graph analytics languages like Green-Marl [8], the algorithm language we use in this paper specifies basic graph traversals as **for** loops over *vertices* and *neighbors* sets. We interleave loops using Rule 7 to achieve concurrent and instruction-synchronous iterations in the active program instances. As explained before and illustrated in Figure 2c, this generic loop interleaving does not make any assumptions about the iterated data or its layout, so each program instance executing the loop may process a different data element. It is, however, more cache-friendly for all instances to process the same data element synchronously.

While enabling element-synchronous iteration on arbitrary collections is beyond the scope of this paper, structure traversals in multi-snapshot graphs allow us two important additional assumptions: First, active program instances traverse collections that are part of the graph's structure and, thus, *very similar*. This is based on the observation that the differences between graph snapshots are only minor compared to the graph's size. Second, we assume that the multi-snapshot graph representation allows efficient traversals of all graph elements as well as determining the snapshots in which they exist. Leveraging these assumptions, we rewrite graph structure iterations in interleaved programs to be graph-synchronous.

To that end we introduce specialized *collect*$_*$ functions for vertex, neighbor and property **for** loops. Consider the following *collect*$_*$ for a graph-synchronous *neighbors* loop which yields all neighbors of v that exist for any program instance:

$$collect_*(G_*.neighbors_*(v), A) = \bigcup_{i \in A} G_*.neighbors_i(v)$$

When the **for** loop iterates over the neighbors set, the same neighbor is processed by all program instances. Because some neighbors do not exist in all instances we additionally define the loop variable's active set for each neighbor:

$$\begin{aligned} n \;&\in\; collect_*(G_*.neighbors_*(v), A) \\ \Rightarrow\; \mathcal{A}(n) \;&=\; \{\, i \mid i \in A \wedge n \in G_*.neighbors_i(v)\} \end{aligned}$$

The active set ensures that even though all neighbors are iterated, they are only visible to program instances for which they actually exist. As per our second assumption, the active set can be efficiently derived from the graph data structure.

Applying optimized graph iteration *collect*$_*$ functions to an interleaved variant of Algorithm 3, the *graph-synchronous* Algorithm 4 can be automatically derived. Note that in this algorithm, the loop variables v and n do not have the $*$ subscript. This signifies that they contain the same value for all program instances, which is the key to graph synchronicity.

Algorithm 4 Graph-synchronous interleaved variant of Algorithm 3.

```
1: fun $SumNeighbors_*(S,\ G_*)$
2:     for $v \in collect_*(G_*.vertices_*(),\ S \cap \Phi_1)$ do
3:         $S_v := S \cap \mathcal{A}(v)$
4:         $R_*[v] := eval_*(0,\ S_v \cap \Phi_2)$
5:         for $n \in collect_*(G_*.neighbors_*(v),\ S_v \cap \Phi_2)$ do
6:             $R_*[v] := eval_*(R[v] + P[n],\ S_v \cap \mathcal{A}(n) \cap \Phi_3)$
7:     return $R_*$
```

Φ_1 through Φ_3 denote dynamic constraints that were introduced by applications of the function and loop rules; e.g., $\Phi_1 = \mathcal{A}(SumNeighbors_*)$. However, as the original algorithm does not contain **break** statements, and because the **return** statement is unconditional, all dynamic constraint evaluations can be automatically optimized away, such that $\Phi_1 = \Phi_2 = \Phi_3 = S$.

Graph-synchronous loops over vertices, neighbors and properties greatly improve the algorithm's temporal and spatial access locality because all program instances access share their data accesses. This property is even preserved in nested loops, distinguishing SAMS from existing approaches [7,23].

4.3 Locality-optimized Data Layout

Interleaving and graph-synchronous execution greatly improve algorithms' temporal locality for accesses to the graph structure and to immutable data, i.e., values that are the same for all analyzed snapshots. An example of this is the global property P in Algorithm 4. Because P is *immutable* it is equal for all program instances; hence, in a graph-synchronous traversal the vertex v's property value can be accessed at the same memory location $P[v]$ for all program instances, greatly improving the accesses' locality.

In contrast, mutable local variables like R_* in Algorithm 4 require changes to the algorithms' data layout to fully benefit from interleaving and graph-synchronous execution. As our transformations cannot make assumptions about the equality of R_*'s values in two different program instances $i, j \in I$, it must allocate R_* for all of them. When the algorithm's original data layout is applied, a separate block of memory R_i is allocated for each program instance i. This is especially problematic for local properties because in this layout there is no spatial locality between a vertex v's property values $R_i[v]$ and $R_j[v]$ in two program instances. Consequently, the temporal locality achieved by graph-synchronous execution cannot be leveraged.

We improve local variables' spatial locality by collocating their values in all program instances. For scalars we allocate all instances' variables as arrays with $|I|$ elements. For properties, instead of allocating a separate block of memory R_i for each instance i, we allocate R_* as one property that stores the instances' values as arrays:

$$R_*[v] = [\ R_{M[0]}[v],\ \ldots,\ R_{M[|I|-1]}[v]\ ]$$

Here, $M\colon \{0,\ \ldots,\ |I|-1\} \to I$ is a bijective mapping of dense numbers to program instances. Using this data layout, graph-synchronous accesses to $R_*[v]$ have spatial locality and, thus, take advantage of the CPU's caches and prefetchers. Furthermore, the collocated layout allows leveraging modern CPUs' wide SIMD instructions to execute interleaved statements for multiple active program instances at the same time.

Note that independent of our data layout, SAMS algorithms' memory consumption for mutable local properties grows linearly with the number of concurrently executed program instances.

4.4 Further Optimizations Opportunities

Interleaving, graph-synchronous traversal, and our optimized data layout transform graph algorithms to greatly improve their efficiency in multi-snapshot analyses. While our automatic transformation changes how the algorithm is executed, it fully preserves its original semantics. For a valid transformation, as described in Section 4.1.8, it is, however, *only* necessary to preserve the algorithm's *externally observable semantics*, especially its result. By leveraging this observation, specialized multi-snapshot graph traversal algorithms and data structures can be designed that use optimizations which are beyond the scope of automatic transformations. In the following we sketch two examples how traversals and data structures can be optimized for multi-snapshot analytics. We envision that multi-snapshot-optimized variants of common algorithms and data structures are part of graph analytics frameworks that leverage SAMS, but they can be provided by algorithm implementers and users as well.

4.4.1 Complex Traversal: BFS

By applying our SAMS transformations, simple textbook BFSs become efficient multi-snapshot BFSs. State-of-the-art BFS algorithms, however, collect heuristics and switch between specialized traversal variants depending on the BFS phase [2]. When such BFSs are automatically transformed, their heuristics are collected separately in each program instance. As a result, the instances independently decide on their locally optimal traversal variant, which can lead to diverging execution paths among the program instances.

We propose that *SAMS-optimized BFSs* should collect aggregated statistics for all program instances and decide on a global traversal variant, similar to [19]. Our experiments show that this greatly improves multi-snapshot BFSs' data access and computation sharing potential, and, thus, improves their performance.

4.4.2 Complex Data Structure: Stack

Consider a simple stack of vertices that stores its entries in an array and keeps the top element's index. The automatically transformed SAMS variant of this vertex stack stores its entries in a locality-optimized array of arrays. As the top element indexes may differ between program instances, *pop* operations are not guaranteed to result in the same vertex for all instances. Thus, popped vertices cannot take advantage of graph-synchronous execution.

We propose a *SAMS-optimized stack* that fosters graph-synchronous execution by ensuring that all active program instances process the same value. Our SAMS vertex stack comprises an entry array E that contains the actual vertices, an array B of bitsets which indicate for which program instances each vertex was pushed onto the stack, and a top element index t that is shared among all instances. We map sets of active instances to bitsets as described in [19].

Pushing a vertex v onto the stack increments t, writes v to $E[t]$ and stores the currently active instances in $B[t]$. Note that v is only stored once no matter how many instances are active. In case the vertex v to be pushed is equal to the current top of the stack and the top value's active bitset does not contain any instance for which v must be pushed, i.e., $E[t] = v \wedge B[t] \cap A = \varnothing$, no new value needs to be added. Instead, the new instances can be added to the bitset of the stack's top element, $B[t] := B[t] \cup A$, thereby improving the potential for future graph-synchronous execution.

The *pop* operation in the active instances A scans B from $i{:=}t$ downward to the first entry with $B[i] \cap A \neq \varnothing$, i.e., the top entry that was pushed by any of the active instances. It returns $E[i]$ for all instances $B[i] \cap A$ and sets $B[i] := B[i] \setminus A$. If $B[i] = \varnothing$ the stack removes the entry and updates t.

Our SAMS-optimized vertex stack enables shared graph-synchronous computation and data accesses, and can significantly reduce memory consumption compared to an automatically transformed stack. Its principles—storing the actual values only once and leveraging bitsets to track in which instances they are valid—can be applied to create SAMS-optimized variants of other common data structures. Examples include priority queues, e.g., for Dijkstra's algorithm, or sets, e.g. for Brandes' betweenness centrality algorithm.

5. EVALUATION

In this section we evaluate the efficiency and scaling behavior of SAMS-transformed multi-snapshot graph algorithms. After describing our experimental setup, we evaluate the influence of different parameters on SAMS's speedup over snapshot-at-a-time execution. Specifically, we look at the number of concurrently processed graph snapshots, the total number of analyzed snapshots, and their similarity. Furthermore, we compare SAMS against two competitors.

5.1 Experiment Setup

We implemented the evaluated six algorithms as standalone single-machine C++14 programs and compiled them with GCC 5.2.1. They are built using the primitives and structures elaborated in Section 4, and transformed to their SAMS variants as described in the same section.

Similar to state-of-the-art graph analytics systems [5, 8], we store graphs in the compressed sparse row (CSR) format. To model the graphs' temporal dimension, we store vertex and edge creation times directly within the CSR.

We ran all experiments on a dual-socket machine with two Intel Xeon E5-2660 v2 CPUs having 20 logical threads at 2.2GHz and 256GB of main memory. The system used Ubuntu Linux 15.10 with kernel 4.2.

5.1.1 Algorithms

To evaluate SAMS, we chose six common graph analytics algorithms that are representatives of different graph access patterns. The *pull-based PageRank* algorithm has a simple, predictable memory access pattern in which the in-neighbor lists of all vertices are traversed and a neighbor property is aggregated. We run it for 20 iterations. *Triangle counting* traverses each vertex v's neighbors n and intersects the neighbor lists of v and n. It represents more complex structure-only graph traversal. For triangle counting, we implicitly undirected all graphs. Breadth-first traversal (BFS) traverses the graph with increasing distance from the source and, thus, randomly accesses vertices' neighbors. We run *4-hop BFSs* from 40 deterministically random selected sources; these traversals stop once all vertices that are reachable within four hops of the source were visited. To show the performance of unbounded BFSs we calculate each vertex's *closeness centrality* value—its average shortest distance to all other vertices [20]. Depth-first traversal (DFS) traverses long paths of previously-unvisited vertices in the graph. In contrast to BFS, a DFS from a source v can follow significantly different paths in two graph snapshots G_{t_1} and G_{t_2}. We run *10-hop DFSs* from 40 deterministically random selected sources. For unbounded DFS traversals we experimented with *Tarjan's strongly connected components algorithm* (SCC). We run SCC only on directed graphs because undirected graphs' SCCs are equal to their weakly connected components, for which significantly more efficient algorithms exist.

Our implementations of PageRank, triangle counting and closeness centrality are parallelized. The BFS, DFS and strongly connected components implementations are serial.

Table 2: Properties of the evaluated data sets.

Graph	Vertices (k)	Edges (k)	⌀ Degree	Diameter	⌀ Diameter	Directed	Temporal
LDBC 1	34.4	1,010.6	29.4	5	2.8		X
LDBC 10	226.9	10,141.4	44.7	6	3.0		X
LDBC 100	1,611.9	101,747.9	63.1	6	3.2		X
Baidu	2,753.2	17,643.7	6.4	24	6.6	X	
Citeseer	384.4	1,751.5	4.6	70	18.6	X	
Mailinglist	27.9	1,014.1	36.3	112	7.2	X	X
Wiki Talk	2,502.0	5,021.4	2.0	14	5.0	X	
Wikipedia	1,870.7	39,953.1	21.4	376	4.6	X	X

5.1.2 Datasets

We evaluate SAMS using multiple real-world graphs as well as synthetic graphs of various sizes. Because of their predominance in practice, we focus on small-world networks. Table 2 shows the evaluated graphs and their properties: vertex and edge count, average degree, full and average diameter, and whether they are directed and temporal.

The synthetic Linked Database Counsil (LDBC) graphs are designed to resemble a social network's friendship graph [9]. We generated them at different scales using the LDBC generator[2], version 0.2.6. Although LDBC graphs are undirected, in our SCC experiments we use them as directed graphs by not generating the implicit back edges to show the algorithm's graph size scalability. All real-world graph datasets were obtained from the KONECT repository [11].

Unless stated differently, we evaluate all algorithms for 512 distinct graph snapshots, where the first snapshot comprises 80% of all edges and their incident vertices. We add the remaining edges and vertices in the subsequent 511 snapshots. For temporal datasets, the subsequent snapshots span equal periods of time but may contain varying edge counts. For

[2]`https://github.com/ldbc/ldbc_snb_datagen`

non-temporal graphs, we deterministically randomize the edges' order and create the subsequent snapshots with equal edge counts. We base our assumption of 20% *change rate*, i.e., edges added after the first snapshot, on multiple social networks' published growth averages per year.

5.1.3 Competitors

We compare our SAMS-transformed algorithms against the respective original non-SAMS algorithms, and variants in Chronos's strategy and a commercial graph analytics system.

Chronos [7] is a graph analytics system that is designed to execute scatter/gather algorithms on multiple snapshots. In contrast to SAMS's algorithm-centric approach, Chronos executes vertex or edge-centric algorithms. This simplifies processing algorithms vertex or edge-synchronously without applying complex transformation rules. As we were not able to obtain the source or a binary of Chronos, we implemented three multi-snapshot scatter/gather algorithms in the authors' proposed design, using the same language and compiler as for our algorithms. We compare the performance of SAMS and our implementation of *Chronos's strategy* in Section 5.3.

Furthermore, we implemented multi-snapshot analyses for PageRank and triangle counting in a commercial in-memory graph analytics system. While this system showed perfectly linear runtime scaling with the number of analyzed snapshots, its absolute runtimes were not competitive. For example, we ran the experiment from Section 5.3, measuring 1024 snapshot computations, on the much smaller LDBC 10 graph. With the commercial system this experiment took 129 minutes for PageRank and 47 minutes for triangle counting as compared to SAMS's 3.8 seconds and 1.5 seconds, respectively. This highlights that existing general purpose graph analytics systems are not suited for multi-snapshot analyses yet. We excluded this system from our further evaluation.

5.2 Degree of Concurrency

An algorithm's SAMS variant is concurrently executed on multiple graph snapshots. We refer to the number of concurrently processed instances as the degree of concurrency ω. While the SAMS transformation works for arbitrary ω, its choice has implications on the maximum possible graph access and computation sharing. The higher ω, the more sharing is potentially possible; however, higher ω also means that a given vertex's algorithm-local properties consume more space in the CPU cache, possibly evicting other important data. To analyze more than ω snapshots, the SAMS algorithm is executed for each batch $A \subseteq S$, $|A| \leq \omega$.

In this section we evaluate how the degree of concurrency ω influences the overall runtime and, thus, the speedup of SAMS execution over traditional snapshot-at-a-time execution. Figure 3 shows the SAMS algorithms' speedup for each evaluated algorithm and varying degrees of concurrency.

Our measurements show that SAMS significantly improves the performance of multi-snapshot analytics for all tested algorithms. Yet, the actual speedup factors are dependent on the algorithms as well as the datasets, as is expected because of their different respective properties. The SAMS *PageRank* algorithm shows speedups of up to one order of magnitude. It already saturates the machine's memory bandwidth for low degrees of concurrency, and the achieved speedup does not change significantly for ω >16. In contrast, SAMS *triangle counting* does not need to access vertex properties. For datasets with a high average degree, it scales almost

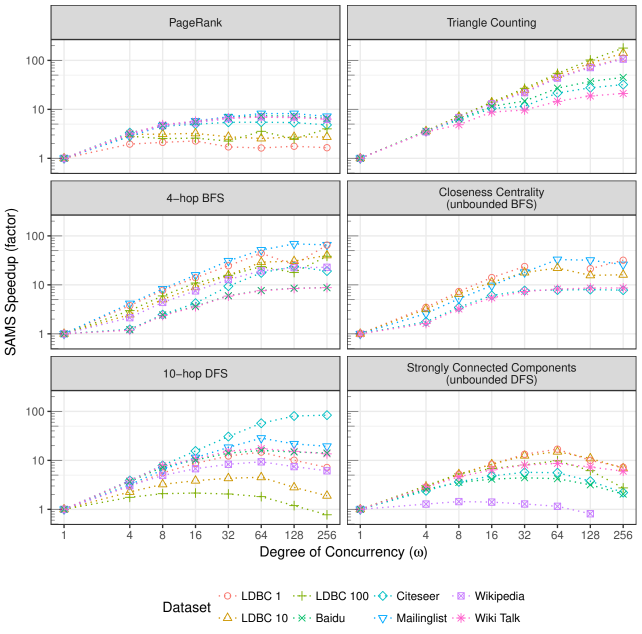


Figure 3: Speedup of SAMS over snapshot-at-a-time execution for varying degrees of concurrency.

linearly with increasing ω, exhibiting a nearly 200x speedup at ω=256 for the LDBC-100 graph. This speedup is caused by SAMS's graph-synchronous execution which avoids redundant computation during common neighbors traversals. The BFS-based algorithms *4-hop BFS* and *closeness centrality* also show good ω-scalability, mainly because SAMS's shared data accesses amortize the cost of memory accesses. For the depth-limited *10-hop DFS*, we see speedups of one to two orders of magnitude because the same paths are traversed in many snapshots, sharing data accesses, and, thus, again amortize the cost of main memory accesses. However, *SCC*'s SAMS variant only exhibits limited speedup. This is because SCC runs unbounded depth-first traversals for which the paths taken by the program instances diverge heavily, so that only very limited data and computation sharing is possible. For the Wikipedia dataset SAMS even caused a slowdown for ω>64 because the cost of SAMS techniques, e.g., the use of data flow instead of control flow (Rule 4), outweighed the initial iterations' performance gains shown in our 10-hop DFS measurements. Such slowdowns can be avoided using a fallback to simple DFS execution when no more sharing is possible, i.e., $|A|=1$.

For ω=64 SAMS shows very good speedups for all tested algorithms. This degree of concurrency is, thus, an advisable choice for practical implementations, should no additional cost model be available.

5.3 Snapshot Count and Competitor

The previous section evaluated SAMS for a fixed number of 512 snapshots. In practical scenarios, other snapshot counts may be of interest. Figure 4 shows the absolute runtimes of executing the PageRank, closeness centrality and triangle counting algorithms on a varying number of snapshots using SAMS and Chronos's strategy. The measurements were done using the LDBC 100 dataset which exhibits typical speedups in our ω-scalability measurements. For SAMS and Chronos's strategy we set the degree of concurrency $4 \leq \omega \leq 256$ such

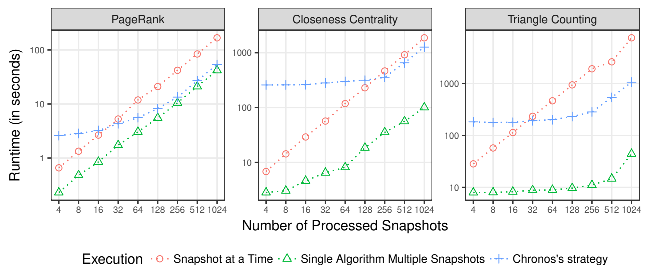


Figure 4: Absolute runtime to process a varying number of snapshots, using the LDBC 100 graph.

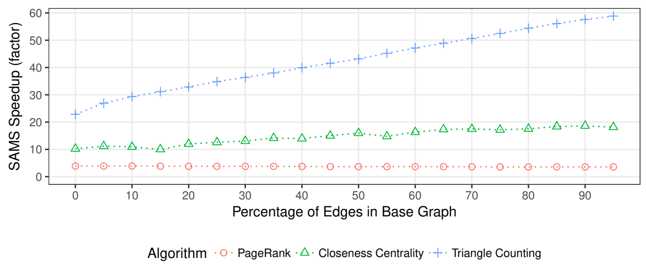


Figure 5: SAMS speedup for varying percentage of edges in the base graph, using the LDBC 100 graph.

that ω is always as close as possible to the actual number of processed snapshots.

Our measurements show that the SAMS *PageRank* is not sensitive to the number of processed snapshots, showing a 3-4× speedup over snapshot-at-a-time execution. Because our original PageRank algorithm is already highly optimized, both the snapshot-at-a-time and the SAMS PageRank perform virtually the same computations. As SAMS improves the execution's locality, its speedup results from amortized random memory accesses during neighbor traversal, which give an almost constant factor. For *triangle counting*, we see that the SAMS variant's speedup over snapshot-wise processing increases with the number of processed snapshots, as more snapshots allow more computation sharing. We see the maximum relative speedup of 173× at 256 snapshots; for more snapshots the speedup remains constant. The reason for this is that our implementation is only designed for $\omega \leq 256$ and runs multiple SAMS batches beyond this number of snapshots. *Closeness centrality* can also leverage the similarities between snapshots well. It shows between 3 and 18× speedup for 4 and 1024 snapshots, respectively.

As pull-based *PageRank* matches the scatter/gather compute model very well, Chronos's strategy exhibits a similar speedup as SAMS when many snapshots are analyzed. For few snapshots its speedup is less pronounced because it must amortize higher fixed costs from the compute model's active vertex management. For the BFS-based *closeness centrality*, Chronos's strategy is significantly slower than the SAMS-transformed algorithm because it must amortize the synchronization cost of message scattering and gathering which is inherent in the vertex-centric approach. This synchronization cost is largely independent of the number of processed snapshots because in Chronos's strategy only one batched message must be sent per vertex and neighbor. We found that for less than 256 processed snapshots the Chronos strategy closeness centrality even exhibits higher absolute runtime than our original snapshot-at-a-time algorithm. The synchronization issue also exists in the *triangle counting* algorithm which scatters each vertex's incident vertices to its neighbors to ensure locality in the gather step. While Chronos's strategy greatly improves the relative performance of scatter/gather triangle counting, its absolute performance is more than an order of magnitude lower than that of SAMS. Furthermore, as a result of temporarily storing the broadcasted neighbor lists, for triangle counting Chronos's strategy inherently has a much bigger memory footprint.

5.4 Snapshot Similarity

SAMS leverages similarities between graph snapshots to avoid redundant data accesses and computations. Thus, it can be expected that the effectiveness of SAMS is proportional to the analyzed snapshots' similarity. To that end, we measured how SAMS's speedup over snapshot-at-a-time execution changes for varying base snapshot sizes between 0%, i.e., the base graph is empty and all edges are added in the snapshots, and 95%, i.e., most edges are in the base graph and only few are added in the snapshots. As we used a fixed snapshot count of 512, the analyzed snapshots are more similar when there are more edges in the base graph. Figure 5 shows our results for fixed $\omega = 64$ for LDBC 100.

We see that SAMS's speedup for PageRank is independent of the snapshots' similarity. This is expected because, as we explained in the previous section, PageRank's snapshot-at-a-time and SAMS variants perform the same number of computations, only differing in their achieved data locality. In contrast, closeness centrality shows higher SAMS speedups for more similar snapshots, confirming our expectation that SAMS's effectiveness is proportional to the snapshots' similarity. For the triangle counting algorithm this is even more pronounced, as it can share more computation among similar snapshots. Our other algorithms also followed this trend.

6. RELATED WORK

Recently, various systems to efficiently store and analyze large graphs have been proposed [5, 8, 15, 18]. In accordance with the majority of analytic graph algorithms, these systems are designed to process **graphs at a single point in time**. SAMS also uses algorithms that are designed to process graphs at a single point in time, but is able to efficiently evaluate them for multiple snapshots of a graph at arbitrary points in time. SAMS does so by means of batched execution.

Existing work shows that in graph analytics, batched execution has great potential to improve the efficiency of specialized algorithms [19] and programming models. In the context of the vertex and edge-centric programming models, Chronos [7] and the very similar PED [23] were proposed for **graph analytics on snapshots**. Vertex and edge-centric algorithms are, however, less expressive [16], hence, they give fewer opportunities to optimize nested computations, data structures and complex graph traversals. We compare SAMS with Chronos in Sections 5.1.3 and 5.3.

Batching has also been studied in the context of **SQL database queries**. Cook and Wiedermann have presented a compiler which groups requests to a remote server to save communication time [4]. Guravannavar and Sudarshan designed a rule-based transformation system for user-defined functions intermixed with SQL [6]. It can hoist recurring SQL queries from **for** loops and expose optimization potential to the database optimizer. SAMS's data access batching and algorithm transformation are conceptually similar to both approaches' SQL query batching. In contrast, SAMS does not rely on subsequent relational optimizers but directly builds the final execution—the transformed algorithm for a fixed batch size. This is beneficial to avoid runtime overhead, because SAMS's unit of batching is a single data access instead of a complex query. Furthermore, SAMS is specifically targeted at the requirements of multi-snapshot graph analytics, which allow leveraging graph specific assumptions that enable, for example, graph-synchronous traversal and our locality-optimized data layout. Thus, SAMS extends existing batching work towards the requirements of graph analytics on modern architectures.

Update-optimized graph database systems like DeltaGraph [10] focus on efficiently storing updates and providing potentially compressed views of the graph at multiple points in time. Similarly, data structures like LLAMA [14] were proposed to efficiently store graph snapshots. SAMS is well-suited to process algorithms on this basis.

In addition, **temporal graph algorithms** that explicitly use the time dimension were proposed in various areas, e.g., shortest paths [22] and centralities [17]. They take into consideration how the graph changes over time and, thus, have different semantics than SAMS-transformed algorithms. Moreover, designing and understanding temporal algorithms poses additional challenges compared to SAMS which automatically applies algorithms to multiple snapshots of temporal and multi-snapshot graphs.

7. CONCLUSION

We showed that Single Algorithm Multiple Snapshot (SAMS) can automatically transform existing graph algorithms such that they share common computation and graph accesses in multi-snapshot analyses. Depending on the algorithm, SAMS can give up to two orders of magnitude speedup over snapshot-at-a-time execution.

8. ACKNOWLEDGMENTS

This research was supported by the German Research Foundation (DFG), Emmy Noether grant GU 1409/2-1, and by the Technical University of Munich - Institute for Advanced Study, funded by the German Excellence Initiative and the European Union Seventh Framework Programme under grant agreement no 291763, co-funded by the European Union. Manuel Then was a recipient of the Oracle External Research Fellowship.

9. REFERENCES

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *SIGACT-SIGPLAN*, pages 177–189, 1983.

[2] S. Beamer, K. Asanović, and D. Patterson. Direction-Optimizing Breadth-First Search. In *SC*, pages 12:1–12:10, 2012.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martı-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[4] W. R. Cook and B. Wiedermann. Remote batch invocation for SQL databases. In *DBPL*, 2011.

[5] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.

[6] R. Guravannavar and S. Sudarshan. Rewriting procedures for batched bindings. *VLDB*, 1(1):1107–1123, 2008.

[7] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen. Chronos: A graph engine for temporal graph analysis. EuroSys, pages 1:1–1:14, 2014.

[8] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: A DSL for easy and efficient graph analysis. In *ASPLOS*, pages 349–362, 2012.

[9] A. Iosup, T. Hegeman, W. Ngai, S. Heldens, A. Prat, T. Manhardt, H. Chafi, M. Capota, N. Sundaram, M. Anderson, et al. LDBC graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *PVLDB*, 9(12):1317–1328, 2016.

[10] U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. In *IEEE ICDE*, pages 997–1008, 2013.

[11] J. Kunegis. KONECT: the Koblenz network collection. In *WWW*, pages 1343–1350, 2013.

[12] J. Leskovec, L. Backstrom, R. Kumar, and A. Tomkins. Microscopic evolution of social networks. In *SIGKDD*, pages 462–470, 2008.

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

[14] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. LLAMA: Efficient graph analytics using large multiversioned arrays. In *ICDE*, pages 363–374. IEEE, April 2015.

[15] 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.

[16] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.*, 48(2):25:1–25:39, Oct. 2015.

[17] R. K. Pan and J. Saramäki. Path lengths, correlations, and centrality in temporal networks. *Physical Review E*, 84(1):016105, 2011.

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

[19] M. Then, M. Kaufmann, F. Chirigati, T.-A. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H. T. Vo. The more the merrier: efficient multi-source graph traversal. *PVLDB*, 8(4):449–460, 2014.

[20] S. Wasserman and K. Faust. *Social network analysis: Methods and applications*, volume 8. Cambridge university press, 1994.

[21] G. Winskel. *The formal semantics of programming languages*. MIT press, 2001.

[22] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu. Path problems in temporal graphs. *PVLDB*, 7(9):721–732, 2014.

[23] W. Xie, Y. Tian, Y. Sismanis, A. Balmin, and P. J. Haas. Dynamic interaction graphs with probabilistic edge decay. In *IEEE ICDE*, pages 1143–1154, April 2015.