Efficient Large-Scale Graph Processing

by

Xiaoen Ju

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2016

Doctoral Committee:

Professor Kang G. Shin, Chair
Professor Peter M. Chen
Professor Hosagrahar V. Jagadish
Hani T. Jamjoom, IBM T.J. Watson Research Center
Associate Professor Carl Lagoze

© Xiaoen Ju 2016
All Rights Reserved

To my parents and my wife

ACKNOWLEDGEMENTS

First and foremost, I thank my parents and my wife for their deep love and tremendous support. It is their encouragement that lights the paths in the countless dark valleys along my Ph.D. journey.

I thank my advisor, Professor Kang. G. Shin, for his insightful advice, extraordinary patience, and generous support. I am truly grateful to him for the wisdom that he shared with me in the past many years. I thank Professor Peter Chen, Professor H.V. Jagadish, and Professor Carl Lagoze for serving on my thesis committee and providing valuable feedback toward improving this thesis. I thank Steve Reger for his administrative support.

It is a great fortune to be in the Real-Time Computing Laboratory (RTCL). I thank all RTCL members who crossed my life path at Ann Arbor. In particular, I am thankful to Xinyu Zhang, who demonstrated to me, in my early years at Ann Arbor, the determination required to survive a Ph.D. journey through his dedication to research. I am grateful to Kai-Yuan Hou, who witnessed to me the indispensable perseverance through her unfailing faith. I thank her for serving as a great forerunner—the only one in almost all our overlapping years at Ann Arbor—in software systems research in RTCL. I thank Xin Hu and Zhigang Chen for their help during my wander in location privacy research. Xin also helped enormously during my job search. I thank Jisoo Yang for his help in Xen-based user application protection research—another detour. I thank Zhaoguang Wang, Caoxie Zhang, Fanjian Jin, Huan Feng, Dongyao Chen, Liang He, and Giovanni Simonini, for all the fun that we shared. I thank members in RTCL's security and systems group, in particular Katharine Chang, Yuanyuan Zeng, Matt Knysz, Buyoung Yun, Seunghyun Choi,

Kassem Fawaz, Arun Ganesan, Kyusuk Han, Yu-Chih Tung, and Kyong Tak Cho, for their feedback.

I am thankful to all my IBM collaborators. I thank Vasileios Pappas, Douglas Freimuth, and Hyoil Kim for their help in virtual machine image deployment research. I thank Dilma Da Silva, Kyung Dong Ryu, and Livio Soares for their help in OpenStack fault resilience research. I am particularly thankful to Livio, for his continuing guidance and support throughout the OpenStack project. I am thankful to Hani Jamjoom and Dan Williams for their help in large-scale graph processing research, which eventually led to this thesis. I would like to give a special thanks to Hani, for his guidance throughout all three graph processing projects and his service on my thesis committee. More importantly, I thank him for his optimism, which brought tremendous momentum into the collaboration.

I treasure the friendships established outside the computer science community, during my years at Ann Arbor. I thank Norma and Roger Verhey, Barb and Norm Fichtenberg, Joyce and Jason Wang, Naomi and Mark Schult, David Chen, and all members in the Ludema fellowship group, in particular Jo and (late) Ken Ludema, and Jihee and Myung Kim, for their caring love, support, guidance, and encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

Efficient Large-Scale Graph Processing

by

Xiaoen Ju

Chair: Kang G. Shin

The abundance of large graphs and the high potential for insight extraction from them have fueled interest in large-scale graph processing systems. Despite significant enhancement in recent years, state-of-the-art large-scale graph processing systems only yield suboptimal performance in common scenarios.

In this thesis, we address three categories of deficiency in graph processing systems. First, in a distributed environment, the performance of graph processing systems is hindered by computation-communication coupling. We present a new programming abstraction called Compute-Sync-Merge that fully decouples computation and inter-host communication, making substantial performance gain.

The second and third categories of deficiency both stem from the lack of version awareness in existing graph processing systems. When multiple versions of a large graph—among which there exists a substantial common subgraph—are processed sequentially, fetching each version as a standalone graph from persistent storage leads to inefficiency. We mitigate such inefficiency by introducing a graph caching layer to a graph processing system, enabling the construction of in-memory graph representation based on cached contents and significantly improving overall system performance.

When multiple versions of a large graph are processed in parallel, maintaining each version as a standalone graph in memory and processing it in isolation from other versions incur memory and computation overheads. We redesign the representation and the processing workflow of a multi-version graph, consolidating duplicated graph states across multiple versions via dynamic version splitting. Our approach improves the overall efficiency of a graph processing system by reducing memory footprint and eliminating computation related to redundant state transitions.

CHAPTER I

Introduction

Large-scale graph processing plays an important role across numerous domains in today's big data analytics. We start this thesis with motivating examples of applying large-scale graph processing to real-world problems. We then discuss state-of-the-art graph processing systems adopting the "think like a vertex" paradigm and identify three categories of deficiency. We highlight our contributions in addressing those categories of deficiency and conclude this chapter with our thesis statement.

1.1 Large-Scale Graph Processing and Its Applications

Graphs are everywhere. With vertices representing entities and properties and edges representing relations and connections among them, graphs are suitable for expressing real-world data in numerous domains. Such real-world graphs are usually large, consisting of millions of vertices and billions of edges.

With the ubiquity of large graphs comes the broad application of large-scale graph processing. For instance, web pages can be classified using label propagation in a web graph, where vertices represent web pages and edges represent overlapping keywords [23]. New movies are recommended to users according to their interests. Such personalized recommendation can be achieved by performing matrix factorization over a graph capturing existing user-movie rating information [5, 6]. Graph-based outlier detection can be applied

to intrusion detection and spam detection [68]. Influential and popular users in a social network graph can be identified by executing a ranking algorithm, such as PageRank [13], over the graph [38]. Graph-based clustering algorithms have proven useful for image labeling [70] and bioinformatics, such as clustering gene expression data [76, 80].

When multiple graphs share a substantial subgraph, they can be considered multiple related versions of a graph—a multi-version graph. A prominent example of a multi-version graph is a collection of snapshots of an evolving graph, such as a social network. Multi-version graph processing is important, because it enables analysis of characteristics embedded across graph versions. In a social network, for example, the change in closeness between users can be studied by executing a shortest path algorithm on temporal snapshots of the network [53]. As another example, the evolving centrality scores of scientific researchers in a co-authorship graph can be captured by executing PageRank on its temporal snapshots [37].

1.2 Think like a Vertex: Abstractions and Systems

Mainstream graph processing systems adopt a "think-like-a-vertex" programming paradigm, proposed by Pregel [46]. Graph algorithms are designed from the perspective of a single vertex, leading to vertex-centric computation. Such computation is expressed via a vertex-centric programming abstraction, for example, in a *compute* function in Pregel or in a series of *gather-apply-scatter* functions in PowerGraph [26].

To execute a graph algorithm, its vertex-centric computation is iteratively applied to all vertices of a graph. Influenced by Pregel, the majority of graph processing systems, such as Giraph [1], Mizan [36], GPS [57], and Pregel+ [78], follow the bulk synchronous parallel (BSP) model [69]. In this model, graph processing progresses in iterations called *supersteps*, in which computation and communication alternate. Computation in a superstep is performed on graph state as viewed at the end of the previous superstep. State update of a vertex within a superstep remains invisible to the rest of the graph until computation

over the entire graph has completed in that superstep. Communication, in the form of inter-vertex messaging or vertex state synchronization, takes place after the completion of computation.

There exist systems that employ asynchronous graph execution, such as Distributed GraphLab [43] and its successor PowerGraph [26], Trinity [60], and GRACE [72]. In asynchronous execution, vertex-centric computation is no longer aligned at superstep boundaries. Vertx state update is immediately visible to subsequent computation, potentially leading to faster convergence of graph state.

1.3 Deficiency in State of the Art

Despite significant advances in large-scale graph processing system design in recent years, such systems often yield suboptimal performance in common scenarios. In this thesis, we investigate three categories of deficiency in state-of-the-art systems: performance penalty due to computation-communication coupling, redundant I/O in graph fetching for sequential multi-version graph processing, and redundant memory state and computation in parallel multi-version graph processing.

1.3.1 Computation-Communication Coupling

Computation and communication are tightly coupled in graph processing systems employing the BSP model. No vertex can proceed to the next iteration of computation until all vertices have been processed in the current iteration and graph states have been synchronized across all hosts. Especially for computationally-light graph algorithms, this coupling of computation and communication incurs significant performance penalty [75].

1.3.2 Redundant Graph Fetching

Multi-version graph processing is an important and common scenario in big data analytics. In such a scenario, each version corresponds to a snapshot of an evolving graph;

a graph processing system iterates over all input versions and applies a user-defined algorithm to them, one at a time. Multi-version graph processing enables the analysis of characteristics embedded across different versions. For example, computing the shortest distance between two users across multiple versions of a social network captures the varying closeness between them [53]. Computing the centrality scores of scientific researchers across multiple versions of a co-authorship graph demonstrates their evolving impact [37].

A key element in multi-version graph processing is efficient *arbitrary local version switching*. Version switching refers to the preparation of the next to-be-processed version after computation completes on the current version. Such a procedure can be arbitrary, because the sequence of to-be-processed versions cannot be predetermined by the underlying system. It is local in that the next version commonly resides in the vicinity of the current version, demonstrating version locality.

Arbitrary local version switching has not been fully addressed before, in particular, from the perspective of the entire multi-version processing workflow. Due to version unawareness, mainstream systems, such as Pregel [46] and PowerGraph [26], perform version switching by discarding the in-memory representation of the current version and loading the next version in its entirety from persistent storage. Such an approach incurs substantial version switching time. Existing multi-version processing systems [21, 37, 39, 45] expedite the version-switching procedure by graph-delta integration. Specifically, the next version is constructed by integrating the current version with the delta representing the difference between the two versions. Albeit efficient, they either lack the support for arbitrary local switching [21, 39], incur high neighbor access penalty during computation [37], or lead to high memory overhead [45].

1.3.3 Redundant Graph State and State Transition

When multiple versions of a graph are processed in parallel, directly extending a version-unaware graph processing system would lead to the instantiation of multiple instances

of the graph processing system, each processing one version. Chronos [31] improves the performance of parallel multi-version graph processing by exploiting locality across versions: when a vertex is scheduled for processing, Chronos executes the vertex-centric algorithm on all versions. Such version-first approach leads to substantial gain with respect to the default vertex-first approach, where all vertices of a graph version are processed before those of a subsequent version.

Neither the version-unaware approach nor Chronos reduces the total memory footprint of a graph processing system or the total computation efforts, leading to redundant in-memory graph state and redundant state transition. In both approaches, all versions processed in parallel need to have their vertex state residing in memory. When the state of a vertex remains the same across multiple versions, however, it is sufficient to maintain only one copy. Regarding computation cost, when a vertex with two versions is activated for computation, if all input states in both versions are identical and the graph algorithm is functional and deterministic, then only one version needs to be processed. The result can be shared between the two versions, saving the effort for another round of computation that would lead to the identical state transition for another version.

1.4 Contributions of the Thesis

In this thesis, we address the above three categories of deficiency in state-of-the-art graph processing systems.

1.4.1 Computation-Communication Decoupling with Compute-Sync-Merge

Intuitively, if graph processing systems can fully decouple computation from communication, they should achieve higher performance because they can better pack communication and computation. We argue that fully decoupling computation from communication can be achieved; it requires (*i*) restricted access to only local state during computation and (*ii*) independence of inter-host communication from computation. We call the combination

of both conditions *local sufficiency*. Conceptually, local sufficiency allows all vertices to always make progress without blocking on input from remote vertices (i.e., those residing on remote hosts). Because all vertices are executing in parallel, local sufficiency can introduce state inconsistency in two ways: (1) vertices might make progress using incomplete input, and (2) replicated vertices might make progress—in parallel—on different sets of inputs. As we will show in the paper, resolving both types of inconsistencies can be done efficiently.

Local sufficiency is not efficiently supported by today's state of the art systems. Synchronous systems, by design, do not support local sufficiency due to their intrinsic computation-communication coupling. Even systems that implement asynchronous execution only partially achieve local sufficiency. For example, PowerGraph [26]'s asynchronous mode satisfies local sufficiency by distributed scheduling. If a vertex-centric function uses remote state, then it will not be marked as ready for execution until its remote input becomes locally available after state propagation. Thus, the function itself is not locally sufficient. Furthermore, the scheduling overhead can be substantial. GiraphUC [29] avoids such a cost by concentrating computation on master vertex replicas, efficiently supporting local sufficiency for edge-cut partitioning. But this approach does not support vertex-cut and thus cannot benefit from the latter's balanced workload distribution.

Towards efficient support for local sufficiency, we set two design goals. The first goal is to activate vertex-centric computation on all vertex replicas, enabling each replica to independently update its local state. This relaxed consistency model would support vertex-cut and enable fast local state propagation without inter-host coordination. The second goal is to enforce local sufficiency at the programming abstraction level. This would eliminate any related coordination overhead at the system level. Additionally, any inconsistency would be resolved by user-defined functions, which are coordinated across all hosts to achieve globally-consistent state upon convergence.

Following these design choices, we introduce a new programming abstraction called

Compute-Sync-Merge (CSM). The Compute abstraction (*Compute* for short) defines locally-sufficient computation, which is iteratively and independently applied to all vertex replicas on each host. Local sufficiency is enforced by the abstraction. *Compute* has access only to local input state. The Sync and Merge abstractions (referred to as *Sync* and *Merge* henceforth) coordinate the execution on all hosts. The former is in charge of state propagation and the latter is responsible for the merging of remote updates with local state. Together, they resolve the inconsistency caused by locally-sufficient computation.

We demonstrate the expressiveness and simplicity of CSM by implementing several widely-used single-phase algorithms, such as PageRank, single-source shortest path (SSSP), and weakly connected component. The CSM abstraction also provides a new dimension for designing efficient locally-sufficient multi-phase graph algorithms. In general, multi-phase algorithms limit the performance gains of locally-sufficient computation due to global synchronization at phase boundaries [29]. CSM, however, enables the design of multi-phase algorithms in which (*i*) locally-sufficient computation freely proceeds beyond phase boundaries and (*ii*) conflicting state due to computation with local input is resolved in *Sync* and *Merge*. We exemplify such use of CSM with an efficient new design of a multi-phase maximal bipartite matching algorithm.

We have fully implemented *Hieroglyph*, a graph processing system supporting CSM on top of PowerLyra [19]. We extend PowerGraph's gather-apply-scatter (GAS) abstraction [26] to realize the CSM abstraction, augmenting the portability of GAS-based algorithms to CSM. Experiments with real-world graphs show that Hieroglyph consistently and significantly outperforms state of the art systems. It outperforms PowerGraph and PowerLyra by up to 22x and GiraphUC by up to 53x, achieving a median speedup of 3.5x and an average speedup of 6x among all algorithm-dataset combinations in our evaluation.

1.4.2 Sequential Mulit-Version Processing with Graph Cache

Towards efficient support for arbitrary local version switching, a system must balance contradicting requirements among three design dimensions: *extensibility*, *compactness*, and *neighbor access efficiency*. Extensibility refers to the easiness of creating a new graph version by extending the current one. Compactness refers to the memory overhead related to version-switching support. Neighbor access efficiency refers to the speed of neighbor access by a graph computing engine.

We present *Version Traveler (VT)* [33], a multi-version graph processing system enabling fast arbitrary local version switching. From a holistic view, VT balances the requirements in all three dimensions of the design space. VT consists of two novel components: (*i*) a *hybrid-compressed-sparse-row (CSR) graph* supporting fast delta integration while preserving compactness and neighbor access speed during graph computation, and (*ii*) a *version delta cache* that stores the deltas in an easy-to-integrate and compact format. Conceptually, the hybrid-CSR graph represents the common subgraph shared among multiple versions in the CSR format. As a result, the subgraph is compactly stored in memory and yields high neighbor access speed—both known advantages of the CSR format. Differences among versions are absorbed by a hierarchical vector-of-vectors (VoV) representation and placed in the delta cache, leading to high version-switching speed thanks to its ability to overcome CSR's lack of extensibility.

We have implemented Version Traveler inside PowerGraph [26] by augmenting its graph representation layer with VT's hybrid-CSR graph and version delta cache. Our evaluation with realistic graphs shows that VT significantly outperforms PowerGraph in multi-version processing: VT runs 23x faster with a mere 15% memory overhead. VT also outperforms designs proposed in state-of-the-art multi-version processing systems, such as log delta, bitmap delta, and multi-version-array, achieving up to 90% improvement when jointly considering processing time and resource consumption.

1.4.3 Computation and Memory State Sharing with Kairos

To address the deficiency related to redundant in-memory graph state and state transition, we propose Kairos, a new graph processing system that provides system support for parallel multi-versioned graph processing. Built on PowerGraph, Kairos can execute a vertex-centric graph algorithm expressed in the gather-apply-scatter (GAS) model. When multiple versions of one graph are specified as input, Kairos automatically and efficiently executes the same algorithm in parallel over the specified versions, exploring and exploiting sharing opportunities along with the progress of the execution. Specifically, Kairos consolidates vertex state identical across multiple versions, maintains one copy per shared state, performs one state transition for each shared state in an iteration, and accomodates new vertex state along the course of execution, for example, when previously shared state between two versions begins to diverge. We have evaluated Kairos across several algorithms over large data sets. Our results show that Kairos can outperform PowerGraph by up to 11.64x when processing 20 versions of realistic large graphs in parallel.

1.5 Thesis Statement

Improving the performance of large-scale graph processing systems is of vital importance. In this thesis, we design and implement three innovative graph processing systems, significantly enhancing state of the art by (*i*) improving the computation stage performance of single-version graph processing by decoupling computation and communication, (*ii*) improving the graph fetching performance of sequential multi-version graph processing via graph state caching, and (*iii*) improving the efficiency of the computation stage of parallel multi-version graph processing by consolidating graph state and eliminating redundant state transition.

CHAPTER II

Hieroglyph: Locally-Sufficient Graph Processing via Compute-Sync-Merge

This chapter starts with a demonstration of the performance penalty related to computation-communication coupling and edge-cut partitioning with respect to vertex-cut, justifying the need for a system capable to decouple computation and communication over vertex-cut partitioning. We then introduce the CSM abstraction in Section 2.2 and present Hieroglyph, a CSM-compliant system, in Section 2.3. Section 2.4 demonstrates Hieroglyph's superiority by comparing its performance with three state-of-the-art systems. Related work is discussed in Section 2.5. We then conclude the chapter in Section 2.6.

2.1 Deficiency of Existing Execution Modes

Synchronous Model. In a synchronous model, vertex-centric computation proceeds in supersteps. Computation and communication iterations alternate. Such coupling can cause a significant performance penalty. Figure 2.1a shows the instantaneous idle time[1] due to communication when running SSSP on a Twitter followers graph with PowerGraph. Along the course of the execution, the idle time remains substantial, indicating considerable blocking of computation due to communication.

[1]The instantaneous idle time is the percentage of execution time that is idle at time t. That is, if in a time interval $(t, t+dt)$, the total idle time is $p\,dt$, then the instantaneous idle time at time t is p.

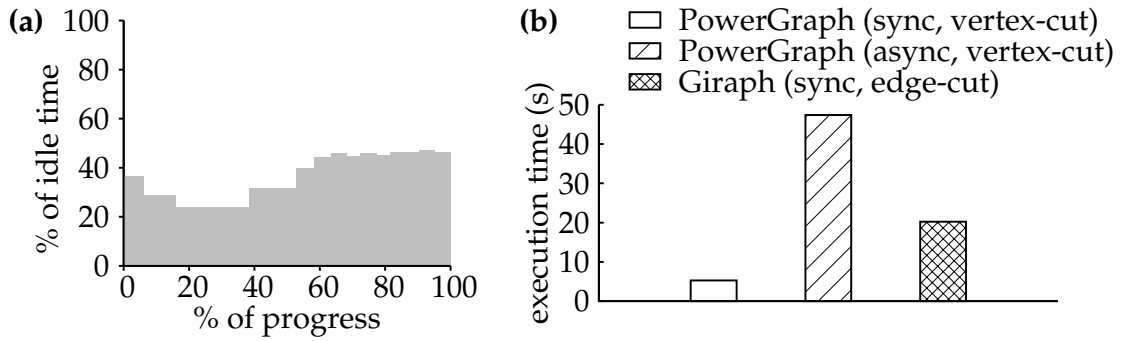

Figure 2.1: Deficiency when executing single-source shortest path (SSSP) over a Twitter followers graph with PowerGraph and Giraph on 16 Amazon EC2 c3.8xlarge instances. (a) Instantaneous idle time (in gray) of PowerGraph. (b) Comparison of execution modes and graph partitioning.

Asynchronous Model. Asynchronous execution [26] decouples computation and communication at the vertex boundary: per-vertex computation-communication tasks are no longer aligned by supersteps and can be independently performed, improving the computation-communication interleaving. Such cross-vertex computation-communication decoupling partially satisfies local sufficiency. For a given vertex, however, the coupling still exists, albeit hidden behind the scheduling of the processing system. Regarding performance, the scheduling overhead, along with the communication deficiency due to the lack of message batching and increased locking overhead, causes asynchronous execution to underperform synchronous execution for several common algorithms, despite the former's faster convergence [42]. For example, Figure 2.1b shows that PowerGraph's synchronous execution significantly outperforms its asynchronous mode, when running SSSP over the Twitter graph.

GiraphUC [29] proposes barrierless asynchronous model, achieving local sufficiency with the observations that (*i*) inter-host communication is much more expensive than intra-host communication and (*ii*) computation can proceed with partial input state propagated via intra-host communication. Remote input state is consumed when it becomes available. GiraphUC supports local sufficiency over edge-cut partitioning. That is, workload related to a vertex—including vertex state update and message passing—must be con-

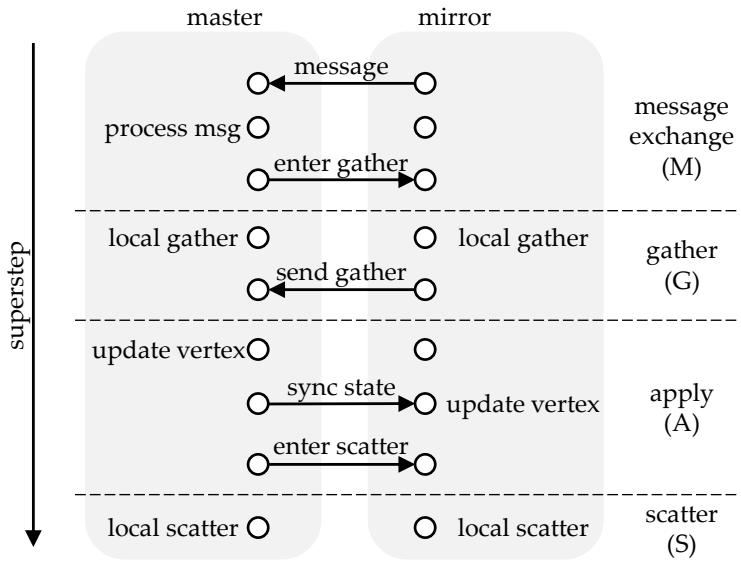

Figure 2.2: The four phases of one superstep in PowerGraph's GAS model, illustrated with a vertex with two replicas

ducted by only one host. Prior work [19, 26] shows that edge-cut partitioning leads to more skewed computation workload and larger memory footprint, compared to vertex-cut. Figure 2.1b compares the performance of PowerGraph synchronous execution with Giraph [1], an open-source implementation of Pregel. Using vertex-cut to produce balanced graph partitions and evenly distribute computation workload, PowerGraph yields a 4x speedup with respect to Giraph, the latter supporting only edge-cut.[2] Given the substantial performance discrepancy between vertex-cut and edge-cut, enabling local sufficiency only for edge-cut would miss an important opportunity for performance improvement.

2.2 Compute-Sync-Merge

In this section, we discuss the Compute-Sync-Merge (CSM) abstraction, including its challenge, design, workflow, consistency model, and expressiveness.

2.2.1 Challenge

Local sufficiency requires (*i*) the restricted access to only local state available at the time of vertex-centric computation and (*ii*) the independence of inter-host communication from local computation. Most existing abstractions cannot fully support local sufficiency, because of the use of remote input state in vertex-centric computation. Such usage is expressed, for example, in the forms of message exchanging in Pregel, vertex state synchronization in Distributed GraphLab [43] and Cyclops' distributed immutable view [18], and distributed gathering/scattering in PowerGraph. GiraphUC achieves local sufficiency for edge-cut but lacks support for vertex-cut.

The unique challenge in supporting local sufficiency on vertex-cut stems from the partial distribution of vertex-centric computation onto multiple hosts, each maintaining a replica of the vertex. Take PowerGraph's GAS abstraction as an example (cf. Figure 2.2). In order to balance the workload, PowerGraph divides vertex-centric computation into three stages: gather, apply, and scatter. The gather and scatter stages are responsible for collecting input state from and propagating updated state to a vertex's neighbors, respectively; they are distributed to all replicas. The state of a vertex is, however, updated only by the master replica in the apply stage. This design leads to computation-communication coupling because of dependencies among vertex replicas during computation. On the one hand, state propagation—in the form of input state in the gather stage and vertex state in the apply stage—is intertwined with computation. On the other hand, coordinated stage transition (i.e., from gather to apply to scatter) across all replicas of a vertex also requires communication. Local barrier and message buffering—techniques proposed in GiraphUC—target state-propagation-related coupling and cannot resolve the additional dependency due to coordinated stage transition.

[2]Differences in systems implementation also contribute to the performance discrepancy.

Table 2.1: CSM Abstraction

compute(D_u, N_u) $\rightarrow$ (D_u^{new}, N_u^{new})
sync(D_u) $\rightarrow$ M_u
merge(D_u, N_u, M_u) $\rightarrow$ (D_u^{new}, N_u^{new})

Table 2.2: Notation

Symbol	Semantics
D_u	data of vertex u
$D_{(u,v)}$	data of edge $u \rightarrow v$
N_u	neighboring states of u, i.e., $\bigcup_{v \in \text{nghr}(u)} \{D_v, D_{(u,v)}\}$
M_u	final sync data for vertex u
I_u^{local}	local sync data for vertex u
I_u^{mirror}	sum of sync data from mirrors of u
I_u^{master}	interim sync data for u at master

2.2.2 Abstraction and Workflow

To achieve local sufficiency in a vertex-cut abstraction, we make two design decisions. First, for vertex-centric computation on each host, we confine the input scope to local state. If such computation consists of multiple stages (e.g., gather-apply-scatter), we activate all stages on all vertex replicas and enable autonomous stage transition without inter-host coordination. This allows computation to proceed at full speed. Second, synchronization of vertex state and merging of local and remote states are needed to ensure the consistency of the final converged graph state across all hosts.

Following this reasoning, we design Compute-Sync-Merge (CSM) (cf. Table 2.1). We discuss CSM along its workflow (cf. Figure 2.3).

A CSM-compliant system maintains two groups of worker threads, one for computation and the other for communication. Locally-sufficient computation progresses in iterations, independently on each host. For an iteration, the computation workers invoke a *Compute* function on all active vertices. A vertex is activated for the next iteration, if it receives messages as a result of the current iteration of computation. The next iteration starts immediately after the completion of the current iteration, without inter-host communication.

The communication workers run in parallel with the computation workers, synchroniz-

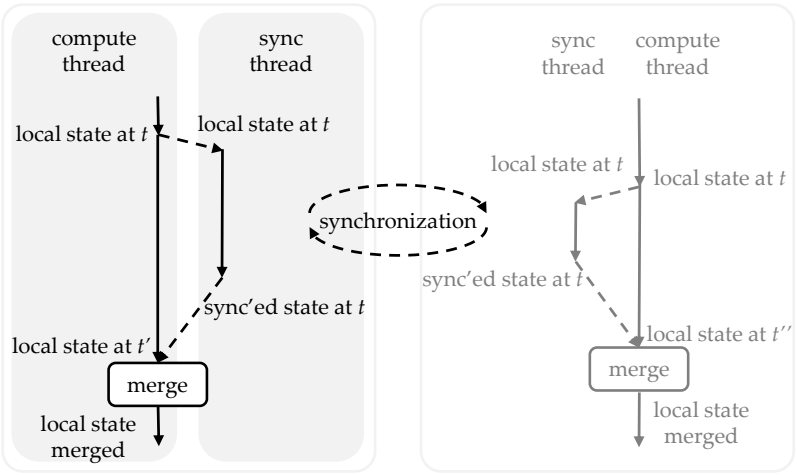

Figure 2.3: Interaction of computation, synchronization, and merging in CSM workflow, illustrated with two hosts (placed left and right). One compute thread and one synchronization thread are shown for each host. Logical time progresses downward.

ing local updates across all vertex replicas with a *Sync* function. State synchronization also progresses in iterations. Contrary to computation iterations, which are locally maintained, synchronization iterations are aligned across all hosts by global barriers. Upon completion of a synchronization iteration, states propagated from remote hosts are merged with the local states used for locally-sufficient computation via a *Merge* function.

2.2.3 Relaxed Consistency

To support local sufficiency on vertex-cut, CSM employs a relaxed consistency model for vertex-centric computation. Specifically, the state of a local replica can be updated independently to other replicas of the same vertex. No ordering constraint is imposed on such updates. Such a model decouples the progress of locally-sufficient computation on each host. Computation is no longer restrained by inter-host communication.

CSM's relaxed consistency is weaker than consistency models in existing synchronous and asynchronous vertex-centric abstractions. Prior work requires that either vertex states be always in sync, or mirrors fall behind masters. CSM, in contrast, allows vertex states to be updated by all replicas. Resolving the inconsistency caused by parallel updates, as

well as the achievement of eventual consistency upon task completion, depends on the collaboration of *Compute*, *Sync*, and *Merge*.

2.2.4 Expressiveness

Algorithms can be expressed by CSM, if they can tolerate transient inconsistency: that is, inconsistent states due to local updates can be resolved by *Sync* and *Merge* and do not affect the correctness of the final converged states. Several common graph algorithms, such as SSSP, weakly connected component, PageRank, maximal bipartite matching, betweenness centrality, and approximate diameter, are tolerant of transient state inconsistency and can be expressed by CSM. We have implemented the first four algorithms.

CSM's expressiveness of inconsistency resolution can lead to better performance, if the gain from local sufficiency outweighs the cost incurred by inconsistency resolution. Such is the case for the four algorithms studied in this paper. Expectedly, with such expressiveness comes the increase of complexity in algorithm design. We evaluate both aspects in Section 2.4.

Algorithms intolerant of local inconsistency can also be expressed in CSM, albeit without the potential performance gain from local sufficiency. This is because stronger consistency can be flexibly reintroduced to the workflow via the three functions in CSM. For example, GAS can be expressed in CSM by (*i*) reducing the collection of local gather states to *Compute*, (*ii*) reducing the propagation of gather states, the generation of the updated master state in the apply function, and the synchronization of the master state, to *Sync*, and (*iii*) reducing the vertex state update in the apply function, as well as local scatter, to *Merge*. GAS algorithms can thus be converted to their corresponding CSM versions.

2.3 Hieroglyph

Hieroglyph is our prototype implementation of a CSM-compliant graph processing system. It is implemented as an integrated component of PowerLyra [19], which is, in turn,

Table 2.3: Hieroglyph's Implementation of CSM

Compute
gather(D_u, $D_{(u,v)}$, D_v) $\rightarrow$ acc
sum(acc left, acc right) $\rightarrow$ acc
apply(D_u, acc) $\rightarrow$ D_u^{new}
scatter(D_u^{new}, $D_{(u,v)}$, D_v) $\rightarrow$ $D_{(u,v)}^{new}$
Sync
sync_switch(D_u) $\rightarrow$ I_u^{local}
sync_sum(I_u^{local} left, I_u^{local} right) $\rightarrow$ I_u^{local}
sync_apply(I_u^{local}, I_u^{mirror}) $\rightarrow$ I_u^{master}
sync_commit(I_u^{local}, I_u^{master}) $\rightarrow$ M_u
Merge
merge_apply(D_u, M_u) $\rightarrow$ D_u^{new}
merge_scatter(D_u^{new}, $D_{(u,v)}$, D_v, M_u) $\rightarrow$ $D_{(u,v)}^{new}$

built on PowerGraph [26]. Hieroglyph extends PowerLyra and PowerGraph's vertex-centric abstraction to support CSM. It implements the CSM workflow in a standalone processing engine, parallel to existing ones in PowerGraph and PowerLyra. It also extends PowerGraph's graph analytics toolkit with algorithms implemented with the CSM abstraction.

2.3.1 Implementation of CSM in Hieroglyph

Hieroglyph decomposes the CSM abstraction into several primitive functions (cf. Table 2.3). We detail this implementation along Hieroglyph's workflow.

Compute. In each locally-sufficient computation iteration (cf. Algorithm 1, Line 3), the computation workers iterate through all active local vertices. Vertex-centric computation is implemented in a GAS style. The reuse of the GAS abstraction in CSM's *Compute* increases the portability of existing GAS-based algorithms to CSM. For each vertex, information regarding its neighbor vertices and edges is accumulated through a *gather-sum* function pair or via a *sum* over messages sent by a previous scatter stage. Such information is then used to update the vertex state via an *apply* function. A *scatter* function concludes the vertex-centric computation by updating neighbor edges according to the new vertex

Algorithm 1 Overall Execution Flow

```
1:  /* executed by compute workers */
2:  while true do
3:    compute an iteration
4:    if sync_worker_state == INACTIVE then
5:      sync_worker_state ← META
6:      resume sync workers for metadata sync
7:    else
8:      if sync_done == false then
9:        continue /* to next iteration of computation */
10:     sync_done ← false
11:     if sync_worker_state == META then
12:       if has_updated_vertices == 0 then
13:         /* all hosts have converged */
14:         terminate computation
15:       else
16:         sync_switch /* prepare sync data */
17:         sync_worker_state ← DATA
18:         resume sync workers for data sync (cf. Algo. 3)
19:     else
20:       sync_worker_state ← INACTIVE
21:       merge
```

state. Hieroglyph's Compute differs from PowerGraph's GAS in that (*i*) all of its three stages are executed on all active vertex replicas and (*ii*) its stage transition is autonomous, without inter-host coordination.

Sync. After an iteration of computation, the computation workers resume the communication workers to perform metadata synchronization (cf. Algorithm 1, Line 6). They then continue with locally-sufficient computation. The goal of metadata synchronization is to achieve consensus among all hosts regarding the set of to-be-synchronized vertices. At the end of the metadata synchronization, if no progress can be made by any host since the last synchronization (cf. Algorithm 1, Line 12), then the computation has completed. Otherwise, computation workers invoke a *sync_switch* function on to-be-synchronized vertices. The purpose of *sync_switch* is to create a standalone copy of the subset of vertex state used for communication, so that subsequent computation can freely proceed, updating vertex state without conflicting with communication. After switching state, the computation

Algorithm 2 Execution Flow: Metadata Synchronization

```
1: /* executed by communication workers */
2: has_updated_vertices ← local_updated_vids.count()
3: ∑^{all hosts} has_updated_vertices
4: /* exclude single-replica (i.e., internal) vertices */
5: updated_vids ← local_updated_vids − internal_vids
6: sync updated_vids across all hosts
7: sync_done ← true
```

Algorithm 3 Execution Flow: Data Synchronization

```
1: /* executed by communication workers */
2: for u in updated_vids do
3:     if u is not master copy then
4:         send I_u^local to master
5: global barrier
6: for u in updated_vids do
7:     if u is master copy then
8:         I_u^master ← sync_apply(I_u^local, I_u^mirror)
9:         send I_u^master to mirror hosts
10: global barrier
11: for u in updated_vids do
12:     M_u ← sync_commit(I_u^local, I_u^master)
13: sync_done ← true
```

workers resume locally-sufficient computation tasks, delegating vertex state synchronization to the communication workers.

The vertex state synchronization is divided into three stages, separated by global barriers (cf. Algorithm 3). In the first stage, all mirror replicas (I_u^{mirror}) are sent to their corresponding master hosts. In the second stage, each host generates intermediate master copy (I_u^{master}) by combining master replicas with received mirror state in a *sync_apply* function and distributes the master copy to the corresponding mirroring hosts. In the third stage, each host creates the final merging state (M_u) for all vertices participating in the current synchronization by combining the local synchronization state and the master copy in a *sync_commit* function.

Merge. Upon communication completion, the computation workers enter the merging stage (cf. Algorithm 1, Line 21). They first use a *merge_apply* function to merge remote

Algorithm 4 SSSP in CSM

Compute

gather(D_u, $D_{(u,v)}$, D_v): no-op
sum(a, b): **return** min(a, b) /* message combiner */
apply(D_u, acc): D_u = min(D_u, acc)
scatter(D_u, $D_{(u,v)}$, D_v): /* to out neighbors */
  if changed(D_u) **and** ($D_u + D_{(u,v)} > D_v$) **then**
    send_msg(v, $D_u + D_{(u,v)}$)

Sync

sync_switch(D_u): $I_u^{local} = D_u$
sync_sum(a, b): **return** min(a, b)
sync_apply(I_u^{local}, I_u^{mirror}): I_u^{master} = min(I_u^{local}, I_u^{mirror})
sync_commit(I_u^{local}, I_u^{master}): $M_u = I_u^{master}$

Merge

merge_apply(D_u, M_u): D_u = min(D_u, M_u)
merge_scatter(D_u, $D_{(u,v)}$, D_v, M_u): /* to out neighbors */
  if changed(D_u) **and** ($D_u + D_{(u,v)} > D_v$) **then**
    send_msg(v, $D_u + D_{(u,v)}$)

state with the local counterpart. Another *merge_scatter* function is then invoked to update neighbor edges according to the newly merged state.

2.3.2 Single-Phase CSM Algorithm Design: SSSP

We exemplify single-phase CSM algorithm design with SSSP (cf. Algorithm 4). Its *Compute* section is expressed similarly to its counterpart in common vertex-centric abstractions [26]. If the shortest distance of a vertex changes due to *apply*, it notifies its neighbors in *scatter*. Messages are combined with the *min* operator, as shown in *sum*.

The *Sync* section involves the propagation of the minimum shortest distance among all replicas. In order to achieve that, *sync_switch* first makes a copy of the local vertex state, which is then propagated from mirror replicas to the master. The shortest distances from mirror replicas are combined with the *min* operator in *sync_sum*. The intermediate master value is the minimum of the shortest distance of the master replica and that of the received mirror replicas. Such a value is then broadcasted to all mirror replicas. *Sync_commit* estab-

lishes the intermediate master value as the final merge value.

As for *Merge*, *merge_apply* and *merge_scatter* have identical functionality to their counterparts in the *Compute* section. Thus, if the local vertex state has a shortest distance that is no larger than the merge value, then the merge value is ignored. This is the case when the current replica contributes the minimum shortest distance to the previous synchronization iteration. It can also happen because locally-sufficient computation further advances the vertex state between the previous synchronization iteration and the merging operation. When merging, the current local vertex state may have become smaller than the minimum shortest distance of all replicas at the time of the previous synchronization. Otherwise, the current local state is larger than the merge value and is thus overwritten by the latter in *merge_apply*. Such update is then propagated to local neighbors in *merge_scatter*.

In summary, SSSP demonstrates the simplicity and elegance of algorithm design using CSM. *Compute* handles locally-sufficient computation and is identical to PowerGraph's GAS implementation, facilitating design reusing. In addition, both the alignment of all vertex replicas to a consistent value in *Sync* and the merge of synchronized consistent state with local vertex state in *Merge* rely on the same message combining and vertex updating logic in the *sum* and *apply* functions of *Compute*.

The CSM implementation of SSSP is guaranteed to converge to the correct final graph state. This is because, if a destination vertex v_d is unreachable from the source vertex v_s, then v_d converges upon algorithm initialization (i.e., ∞). If v_d is reachable, then let n denote the minimum number of vertices along the shortest path(s) from v_s to v_d (inclusive). v_d enters the final converged state within $n-1$ iterations of synchronization: after at most i iterations, vertices i steps away from v_s receive the correct final states, propagated via *scatter* or *merge_scatter*.

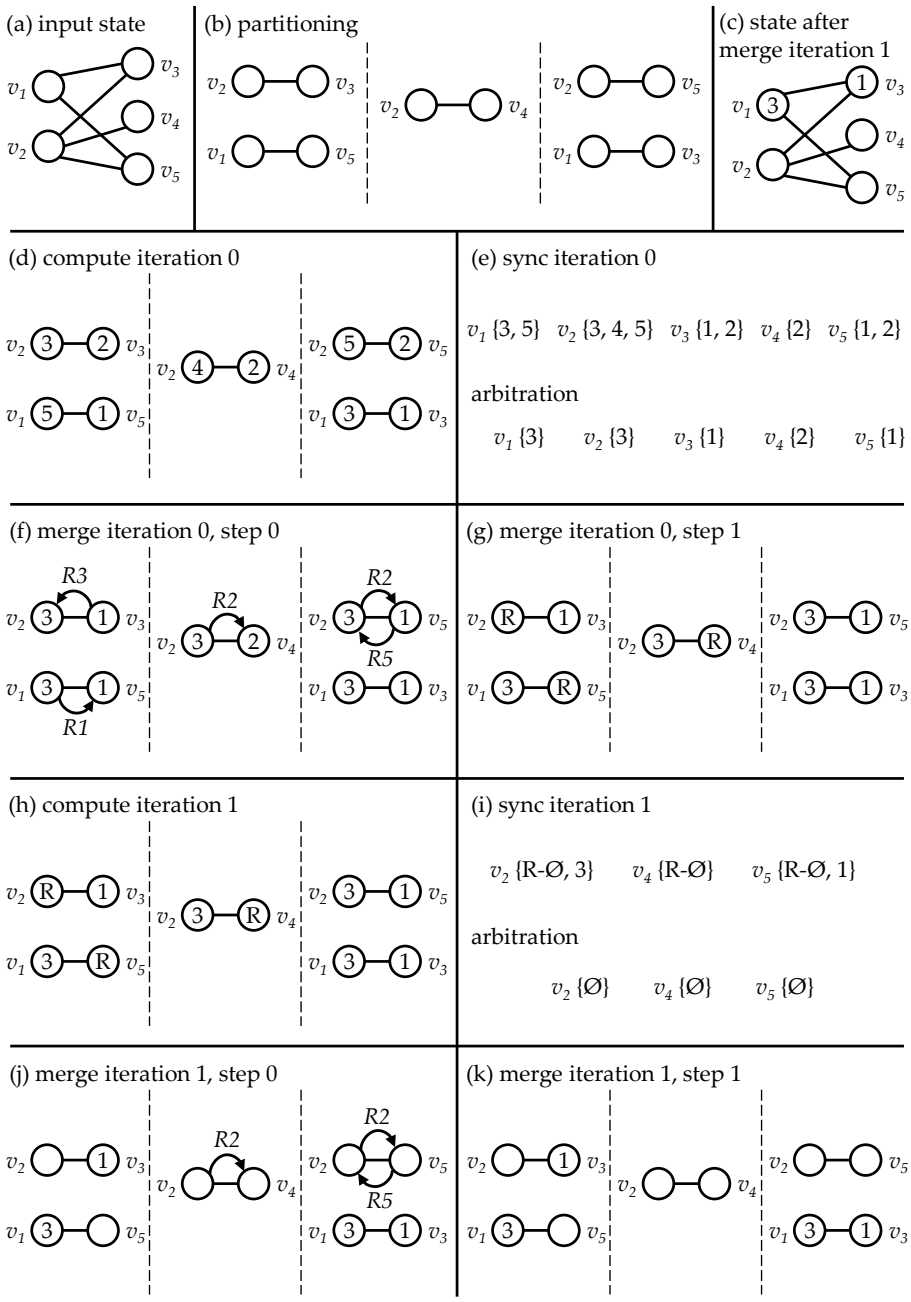

Figure 2.4: Illustration of bipartite matching in CSM. Each circle represents a vertex. The number inside a circle represents the id of the vertex with which the center vertex (represented by the circle) currently matches. A circled "R" represents the revoked state. An arrow associated with "R{vid}" represents a local revocation message with *vid* as its payload.

2.3.3 Mulit-Phase CSM Algorithm Design: Bipartite Matching

Multi-phase algorithms impose new challenges to locally-sufficient computation [29]. Due to the different functionality across computation phases, global synchronization is

generally required to guarantee the correctness of execution. In addition, messages sent from a phase to be processed by the subsequent phase must be hidden until phase switching. Otherwise, they will be combined with messages targeting at the current phase, leading to erroneous results. As a result, locally-sufficient computation is applied only phase by phase in GiraphUC, reducing the performance gain.

In CSM, when (*i*) progress can be made *across* algorithm phases based only on local input state and (*ii*) inconsistent vertex state as a result of cross-phase locally-sufficient computation can be fixed without affecting the correctness of the final converged graph state, locally-sufficient computation can freely proceed across phase boundaries, leading to potentially high-performance algorithm design.

We demonstrate the potential of CSM in multi-phase algorithm design with our implementation of the bipartite matching algorithm. We adapt the four-phase matching algorithm in Pregel [46] for CSM's *Compute*. When applying it to locally-sufficient computation with no inter-host coordination, that algorithm produces a maximal matching on each computing host. Inconsistent final state may occur, nevertheless, in the form of vertices having different matching state across multiple replicas. Such inconsistency is resolved in *Sync* and *Merge* of the CSM version of the matching algorithm.

Bipartite Matching by Example. We use an example to describe bipartite matching in CSM, shown in Figure 2.4. Detailed implementation of the algorithm can be found in Algorithms 5–7. The input graph (cf. Figure 2.4a) has five vertices, two on the left and three on the right, with five edges connecting them. Suppose the graph is partitioned onto three hosts (cf. Figure 2.4b). We follow the execution of the algorithm step by step until the completion of the second iteration of synchronization and merging (shown in Figures 2.4d–2.4k), reaching the state shown in Figure 2.4c. Note that, for bipartite matching, CSM confines the resumption of communication workers to the start of the four-phase matching cycles. The semantics of *one iteration of computation* in CSM's overall execution flow (cf. Algorithm 1, Line 3) thus refers to a cycle of four-phase matching. Intermediate matching state

within a cycle is hidden from communication workers, simplifying the synchronization.

In the first iteration of computation, matching is performed locally (cf. Figure 2.4d). Local state is then synchronized (cf. Figure 2.4e), with the master replica of each vertex accumulating all local matching decisions. Each master replica then uses an arbitration function to resolve conflicting decisions. In our current implementation, each master replica selects the matching decision with the minimum matching vertex id. Such decision is then distributed to all replicas of a vertex and is used as its merge value.

Merge is separated into two steps in bipartite matching. In the first step, merge value obtained from synchronization is used to correct local decisions (cf. Figure 2.4f). When a local decision is corrected, it also sends a revocation message to its previous partner. For example, on the left host, after v_3's local decision of matching with v_2 is overwritten by the merge value v_1, v_3 sends a message to v_2, notifying its revocation of its previous matching decision. In the second step (cf. Figure 2.4g), each vertex receiving revocation messages checks the relevance of those messages. That is, whether the payload of a revocation message matches the current matching decision of the receiving vertex. Irrelevant revocation messages stem from global state synchronization: upon receipt of a revocation message, the vertex's state may have been updated to its globally consistent value. Since the receiving vertex has rematched to a vertex different than the sending vertex, the sender's revocation of a previous matching becomes irrelevant and is ignored. Such are the cases for v_2 and v_5 on the right host, each receiving an irrelevant revocation message from the other. When a relevant revocation message arrives (e.g., v_2 on the left host), the receiving vertex sets its state to revoked, becomes unmatched, and then activates itself for the next iteration of computation.[3]

The second iteration of computation progresses as the first one (cf. Figure 2.4h). Since all revoked vertices have their local neighbors in a matched state, no process can be made. In the subsequent synchronization (cf. Figure 2.4i), local revocations from the previous

[3] A revoked right vertex also needs to activates all its unmatched left neighbors.

merging phase are propagated globally. The revocation precedence adopted in the arbitration logic (cf. Algorithm 6) guarantees that revocation decisions overwrite matched state. For example, regarding v_2, the unmatched state (represented by $\varnothing$) prevails after a message combination with a matching decision of v_3.

The second iteration of merging proceeds as the first one (cf. Figures 2.4j and 2.4k), applying synchronized state and propagating local revocations in the first step and performing or ignoring revocations in the second step. After the merging, the graph reaches the state as shown in Figure 2.4c.

Algorithm Design Details. Algorithms 5–7 illustrate the CSM implementation of the bipartite matching algorithm.

The *Compute* section mirrors the four-phase matching algorithm in Pregel [46]. In Phase 0, each unmatched left vertex sends a matching request to its unmatched right neighbors. In Phase 1, each unmatched right neighbor grants matching to one requesting neighbor (via replying the right neighbor's vertex id) and declines the remaining requests (via replying an empty message). In Phase 2, if a left vertex receives at least one grant message, it then selects one such message, updates its own state to record the matching, and confirms with the matching right vertex (via replying its vertex id). In Phase 3, a right vertex receiving a confirmation message records the matching by updating its own state.

For *Sync*, the key point is to achieve consensus on the state of a vertex. Vertex state is propagated across all replicas. The arbitration follows *revocation precedence*: state from a vertex with a "revoked" flag overwrites that from a normal (i.e., non-revoked) vertex replica. If two vertices have the same "revoked" flag, then they have the same priority and the arbitration is based on their matched state. In our current implementation, when inconsistent local matching decisions occur, we favor the matching with the smallest vertex id. Users are free to implement their own arbitration logic, as long as it guarantees forward progress.

For *Merge*, *merge_apply* is divided into two phases: Phase 0 is in charge of updating

Algorithm 5 Bipartite Matching in CSM: Compute

```
Compute
gather(D_u, D_(u,v), D_v): no-op
sum(a, b): /* merge requesting/granting vid vectors */
  return a.insert(b)
apply(D_u, acc):
  /* randomly select a message */
  if acc.empty() then D_u.msg = MAX
  else D_u.msg = random_select(acc)
  /* check phase */
  if (iteration_counter mod 4) == 2 then
    if D_u.msg != MAX then
      D_u.vid = D_u.msg /* left becomes matched */
    else send_msg(u, ∅) /* keep activated */
  if (iteration_counter mod 4) == 3 then
    if is_right(D_u) then
      D_u.vid = D_u.msg /* right becomes matched */
    else send_msg(u, ∅) /* keep activated */
scatter(D_u, D_(u,v), D_v):
  /* check phase */
  if (iteration_counter mod 4) == 0 then
    /* if left and unmatched, scatter to all neighbors */
    if is_right(v) and (not is_matched(D_v)) then
      send_msg(v, u)
  if (iteration_counter mod 4) == 1 then
    /* scatter to all neighbors */
    if is_left(v) then
      if v == D_u.msg then send_msg(v, u) /* grant */
      else if (not is_matched(D_v)) then
        send_msg(v, ∅) /* decline */
  if (iteration_counter mod 4) == 2 then
    /* scatter to all neighbors */
    if D_u.msg == v then send_msg(v, u) /* confirm */
```

local state with globally consistent state thanks to the arbitration in *Sync*; Phase 1 is for performing local revocation. In Phase 0, the globally consistent state is authoritative, unless it is a normal (i.e., non-revoked) and unmatched state. In that case, it will not be used to overwrite local state, thus preserving progress made by locally-sufficient computation. In other cases, the local state is then aligned with the globally consistent state. If a local matching decision is overturned, then the related vertex sends a revocation message, notifying the

Algorithm 6 Bipartite Matching in CSM: Sync

Sync

sync_switch(D_u): $I_u^{local} = D_u$

sync_sum(a, b): /* arbitration */

  if $a.revoked == b.revoked$ **then**

    /* same revocation precedence, compare vid value */

    if $a.vid$ ¿ $b.vid$ **then** $a.vid = b.vid$

  else if (**not** $a.revoked$) **then** $a = b$ /* diff precedence */

sync_apply(I_u^{local}, I_u^{mirror}):

  I_u^{master} = sync_sum(I_u^{local}, I_u^{mirror})

sync_commit(I_u^{local}, I_u^{master}): $M_u = I_u^{master}$

locally-matched partner vertex.

A vertex enters Phase 1 of *merge_apply* if and only if it receives local revocation messages. Such a vertex then determines whether its received messages are relevant. Recall that a local revocation messages is relevant if its sender is the vertex with which the receiving vertex records a match confirmation. In this case, the receiving vertex resets its matching state to unmatched (i.e., *MAX*) and turns on its "revoked" flag. Such local revocation will be propagated to other replicas in the next iteration of synchronization. Due to the revocation precedence in *Sync*, state propagated from revoked vertex replicas correctly overwrite normal replicas. If, after revocation, a vertex becomes unmatched, then it needs to re-enter the next round of computation. If such a vertex is on the right, it also needs to reactivate all its unmatched left neighbors for the subsequent computation.

Correctness. The CSM implementation of bipartite matching is based on the original Pregel implementation—a randomized maximal matching algorithm. Its correctness is determined by the properties of the final graph state.

Definition II.1. A distributed randomized maximal bipartite matching algorithm is correct if, upon graph convergence, (i) all replicas of a vertex are in the same state; (ii) if a vertex is matched, then it matches with a vertex on the other side of the bipartite graph; (iii) if a vertex v_i matches with another vertex v_j, then v_j matches with v_i; and (iv) if a vertex is unmatched, then all its neighbors on the other side of the graph, if any, are matched.

Algorithm 7 Bipartite Matching in CSM: Merge

```
Merge
merge_apply(D_u, M_u):
  /* check merge phase */
  if (merge_counter mod 2) == 0 then
    if (not M_u.revoked) and (not is_matched(M_u)) then
      return
    D_u.revoked = false
    if D_u.vid != M_u.vid then
      /* local state is out-of-sync, revoke */
      prev = D_u; D_u.vid = M_u.vid
      if is_matched(prev) then
        /* need revocation, msg stored in merge_msg */
        send_revoke_msg(prev.vid, u)
        if not is_matched(D_u) and is_left(u) then
          send_msg(u, ∅) /* re-enter compute */
  else
    if merge_msg.contains(u) then
      /* msg is relevant. perform local revocation */
      D_u.vid = MAX; D_u.revoked = true
      send_msg(u, ∅) /* re-enter compute */
merge_scatter(D_u, D_(u,v), D_v, M_u): /* to all neighbors */
  if is_right(u) and become_revoked(u) and is_left(v)
    and (not is_matched(D_v)) then send_msg(v, ∅)
```

Regarding the four properties of the final graph state, the first addresses general consistency of distributed graph state. The second property is the requirement of bipartite matching. The third property is the consistency requirement of a matching. The fourth property is the requirement of a maximal matching.

Theorem II.2. *The CSM implementation of bipartite matching is correct.*

We prove Theorem II.2 by proving the following lemmas.

Lemma II.3. *Upon graph convergence, all replicas of a vertex are in the same state.*

Proof. The final state of all replicas of a vertex v_i is the same as the state decided by the arbitration logic in the last synchronization iteration k related to v_i. After that iteration of synchronization, there exists no local update to v_i at any host. This is because, any further

local update to a replica of v_i after k, due to either computation or revocation, leads to another synchronization iteration $k+1$ related to v_i, contradicting with the assumption that k is the last synchronization for v_i.

The consistent final state of v_i is thus an outcome of the arbitration logic, which selects one and only one value from all replicas of v_i. □

Lemma II.4. *Upon graph convergence, if a vertex is matched, then it matches with a vertex on the other side of the bipartite graph.*

Proof. All matching decisions are results of local matching decisions, direct (on the same host where the local matching takes place) or indirect (via propagation). Local matching decisions are made according to the original proven-correct bipartite matching algorithm used in Pregel. Left vertices thus only match with right vertices, and vice versa. □

Lemma II.5. *Upon graph convergence, if a vertex v_i matches with another vertex v_j, then v_j matches with v_i.*

Proof. Suppose, in the final graph state, v_i matches with v_j but v_j does not match with v_i.

For v_i, the matching between v_i and v_j stems from a local matching decision made by a host h maintaining the edge connecting v_i and v_j. The last state update of v_i on h must be because of the local matching decision. Note the time when the last update of v_i on h takes place as t.

At t, v_j's local state at h is "matched with v_i," due to the local matching decision. Since v_j's final state is "not matched with v_i," the local state of v_j on h must be updated at t', $t < t'$. At t', v_j sends a local revocation message to v_i, according to the two-phase merging logic (cf. Algorithm 7). Such revocation message must be relevant to v_i, because v_i's state remains "matched with v_j" after t. This relevant revocation message will then cause v_i's local state to change, contradicting with the assumption that v_i's state is final after t. □

Lemma II.6. *Upon graph convergence, if a vertex is unmatched, then all its neighbors on the other side of the graph, if any, are matched.*

Proof. Support v_i and v_j are neighbors on opposite sides of the graph and both unmatched in the final graph state.

There are two cases where v_i and v_j can be both unmatched in the final graph. First, v_i and v_j remain unmatched during the course of the computation. Second, at least one of them becomes matched during the computation but then becomes unmatched due to CSM's revocation logic.

In the first case, there must be a host h maintaining the edge connecting v_i and v_j. As a result, the two vertices cannot remain unmatched on h, according to the proven-correct local matching algorithm.

In the second case, without loss of generality, assume that v_i enters the final unmatched state no earlier than v_j. When v_i enters the final unmatched state, it activates itself and/or v_j on h, according to the two-phase merging logic (cf. Algorithm 7). After the activation, the two remain unmatched on h—an impossible condition according to the proven-correct local matching algorithm. □

According to Lemmas II.3–II.6, we have Theorem II.2 by Definition II.1.

2.3.4 Discussion

Termination Condition. In Hieroglyph, the termination condition is checked against the total number of vertices updated across all hosts since the last synchronization (cf. Algorithm 1, Line 12). An algorithm completes when this number becomes zero.[4] In contrast, in the synchronous mode of PowerGraph and PowerLyra, this condition is checked at the superstep boundaries, owing to the tight coupling between computation and communication. If no progress can be made after an iteration of computation followed by an iteration of communication, then the execution terminates.

[4]This statement holds for single-phase algorithms. It also holds for multi-phase algorithms whose CSM implementations do not require global synchronization, such as our bipartite matching algorithm. For multi-phase algorithms mandating global synchronization at the phase-switching boundaries, this condition marks the end of a phase.

The correctness of Hieroglyph's termination condition derives from the following patterns in the workflow (cf. Algorithm 1): (*i*) one iteration of computation (Line 3) is performed after an iteration of merging (Line 21) and before the termination condition checking (Line 12), and (*ii*) the termination checking, in turn, precedes the subsequent iteration of synchronization (Line 18). If no progress can be made in any host since the last synchronization, after incorporating the remote states in the merging stage and a subsequent iteration of locally-sufficient computation, then no progress is possible. Convergence over the entire graph has thus been achieved.

Computation-Communication Interleaving. In Hieroglyph, to achieve computation-communication decoupling, we use two groups of worker threads: one for computation and the other for communication. PowerGraph and PowerLyra, in contrast, rely on one group of threads to perform both computation and communication.[5]

The two groups of worker threads in Hieroglyph are of an equal size, both equating the number of cores on a computing host. Hieroglyph supports fine-tuning of computation-communication interleaving. On the one hand, in order to expedite the integration of remote vertex state updates into locally-sufficient computation, the computation workers separate vertices into chunks and perform computation one chunk at a time, yielding to the communication workers at the chunk boundaries and thus improving their interleaving.[6] On the other hand, Hieroglyph can be configured to enforce algorithm-specific restrictions on computation-communication interleaving. Our bipartite matching algorithm, for instance, confines the interaction between the two groups of worker threads only to the boundaries of a four-phase computation cycle. It ensures that local matching decisions, instead of intermediate states during locally-sufficient multi-phase computation, are used for synchronization.

[5]Our discussion focuses on threads used by the processing engine layer and does not include those used for background communication.

[6]One iteration of computation over *all* local vertices is performed immediately after an iteration of merging, regardless of chunk configuration, in order to maintain the validity of the termination condition.

Deferred Switching. Deferred switching refers to postponing the preparation of the synchronization state from the beginning of the synchronization to when the state is accessed by the communication workers. Specifically, it delegates the invocation of *sync_switch* from the computation worker (cf. Algorithm 3, Line 16) to the communication workers, before the sending of I_u^{local} in the case of a mirror replica (cf. Algorithm 3, Line 4) and before the invocation of *sync_apply* for a master replica (cf. Algorithm 3, Line 8). In comparison, the original workflow of the computation workers (cf. Algorithm 1) invokes *sync_switch* before the start of the communication, isolating the vertex state used by computation and communication workers and thus guaranteeing lock-free access.

Deferred switching relaxes such isolation, exploiting benign data race to expedite state propagation. For example, in SSSP (cf. Algorithm 4), *sync_switch* involves a copy of the locally-maintained shortest distance. Since this value is monotonically non-increasing, correctness remains intact if functions in *Sync* access a vertex state different from the one that would have been copied by *sync_switch* before the start of a synchronization iteration. It is also beneficial to defer the access of the vertex state during synchronization: the use of an updated state from one host potentially expedites convergence on other hosts. The monotonically non-increasing property, combined with read-only access from the communication workers, justifies benign data race: access to vertex state from both worker groups remains lock-free.

In Hieroglyph, we support both the eager switching design and the deferred switching mode as an optimization.

Local Synchronous/Asynchronous Execution. The CSM abstraction does not define whether locally-sufficient computation should be performed synchronously or asynchronously on each host. Assume PowerGraph GAS for locally-sufficient computation. For an iteration of computation, locally-synchronous execution involves running the gather, apply, and scatter phases each in a dedicated iteration, with a local barrier separating two consecutive phases. Messages produced in the current iteration can be processed only in the subsequent

iteration. In locally-asynchronous execution [29,66,75], on the other hand, the three phases are applied to a vertex as an integrated function. Messages sent from vertices v_i to v_j are processed in the same iteration, if v_j is processed after v_i.

Locally-synchronous execution achieves lock-free access to vertex and edge data. Locally-asynchronous execution, in contrast, has the advantage of fast state propagation. Hieroglyph supports both execution modes and, for locally-asynchronous execution, supports different consistency models, similar to the vertex/edge/full consistency in distributed GraphLab [43].

Fault-Tolerance. Hieroglyph resorts to a checkpoint-based fault-tolerance mechanism [1, 26, 36]. Checkpoints are created by computation workers independently on each host. The checkpoint interval is specified with respect to synchronization iterations, which are globally consistent, and aligned at the beginning of a metadata synchronization stage. Only vertex states and intra-host messages need to be checkpointed. This is because, at the time of checkpoint creation, there exists no state associated with either *Sync* or *Merge*.[7] When a fault occurs, all hosts are rolled back to their most recent checkpoints.

2.4 Evaluation

We compare the performance of Hieroglyph with three state-of-the-art graph processing systems and show the superiority of our proposed CSM abstraction.

2.4.1 Experiment Setup

To evaluate Hieroglyph's performance, we use five realistic datasets (summarized in Table 2.4). Livejournal [7, 40] describes the friendship relation in the LiveJournal social network. Wiki [11, 12] compiles English Wikipedia pages. Twitter [38] captures the "who follows whom" relation in the Twitter social network. Road [25] is the road network of

[7] The next *Sync* stage is pending and the previous *Merge* has completed, with all its related state incorporated into vertex state. Also recall that no inter-host message exists in locally-sufficient computation.

Table 2.4: Graph datasets. Counts in parenthesis are for undirected graphs.

Dataset	$\|V\|$	$\|E\|$
Livejournal	4.8M	69.0M (85.7M)
Wiki	4.2M	101.3M (183.9M)
Twitter	41.7M	1.5B (2.4B)
Road	2.8M	6.8M (6.8M)
Web	118.1M	1.0B (1.7B)

the great lakes area of the United States. Web [11, 12] is a web graph generated by WebBase [24].

We evaluate Hieroglyph with four algorithms: bipartite matching (abbreviated as *Bipart*),[8] weakly connected component (abbreviated as *CC*), PageRank, and SSSP, all common building-block algorithms in graph analytics.

We compare Hieroglyph with PowerGraph [26], PowerLyra [19], and GiraphUC [29]. PowerGraph uses vertex-cut partitioning and features efficient processing of high-degree vertices. PowerLyra extends PowerGraph to support hybrid-cut, enhancing computation efficiency for low-degree vertices. Hieroglyph augments PowerLyra with locally-sufficient computation. Given such relation, regarding both design and implementation, a performance comparison between Hieroglyph and its two predecessors identifies the gain of locally-sufficient computation over vertex-cut. GiraphUC is a vertex-centric graph processing system providing locally-sufficient computation over edge-cut. Despite the discrepancy between GiraphUC and Hieroglyph in terms of implementation details,[9] a comparison between them sheds light on the potential of enabling locally-sufficient computation over vertex-cut partitioning.

All experiments are conducted in a cluster of 16 Amazon EC2 c3.8xlarge instances, each with 32 2.8GHz vCPUs, 60GB memory, and 10 Gbps network connection. All instances run Ubuntu 14.04.2 LTS (Linux 3.13.0-54-generic). PowerGraph, PowerLyra, and

[8] Treating vertices with an even id as left vertices and the rest as right vertices enables the evaluation of Bipart on all five datasets [56].

[9] Such discrepancy includes different vertex-centric abstractions, programming languages, and inter-host communication mechanisms.

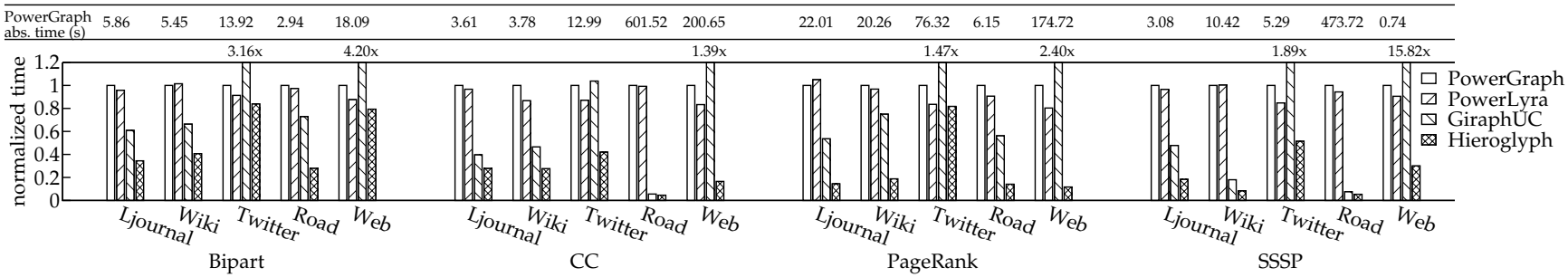

Figure 2.5: Performance comparison. Execution time is normalized to that of PowerGraph. Absolute average execution time of PowerGraph (in seconds) is marked atop each cluster. Normalized execution time of GiraphUC, when exceeding the plotting range, is also marked.

Hieroglyph are compiled with gcc 4.8.2. GiraphUC is implemented on Giraph 1.1.0 and run with Hadoop 1.0.4 and jdk 1.7.0_79. Each data point is the mean of at least three runs. For all experiments, grid vertex-cut is used for PowerGraph by default, hybrid-cut with ginger heuristics for PowerLyra and Hieroglyph, and hash-based edge-cut for GiraphUC. PowerGraph and PowerLyra run in the synchronous mode by default.

2.4.2 Performance

Figure 2.5 shows that Hieroglyph outperforms all the other three systems for all algorithm-dataset combinations in our evaluation. Hieroglyph's speedup varies from 1.02x to 52.50x, with a median speedup of 3.54x and an average speedup of 6.00x.[10]

Comparing against PowerGraph and PowerLyra. In most settings, the performance improvement of Hieroglyph with respect to both PowerGraph and PowerLyra maximizes on Road and minimizes on Twitter. This is because locally-sufficient computation is most effective with respect to synchronous execution, when local state propagation in synchronous execution is severely hindered by global synchronization. Hieroglyph's effectiveness is thus amplified by Road, which has a large diameter and requires numerous supersteps—each concluded with an iteration of global synchronization—for local state propagation in PowerGraph and PowerLyra.[11]

The diminishing performance gap in Twitter can be understood from the perspective of computation-communication balancing. Given its size, Twitter imposes considerable computation workload on each participating host. On the one hand, it improves the computation-communication interleaving in PowerGraph and PowerLyra, effectively reducing the penalty of inter-host communication. On the other hand, it magnifies the computation overhead of Hieroglyph caused by (*i*) repeated per-vertex computation to process asyn-

[10] GiraphUC runs in BSP mode (i.e., reducing to Giraph) in our Bipart measurement. This is because GiraphUC terminates prematurely when executing our Bipart algorithm with locally-sufficient computation. Note, however, that GiraphUC's execution time obtained in BSP mode lower-bounds that with locally-sufficient computation enabled, given the one-superstep-per-phase property of Bipart.

[11] Such pattern conforms to observations in GiraphUC [29].

chronously-delivered input state and (*ii*) the need for resolving inconsistent local state. Indeed, compared to Livejournal and Wiki, we observe an increase in vertex update rate—an indicator of the computation efficiency—for PowerGraph and PowerLyra in the case of Twitter. The update rate for Hieroglyph, however, drops in the case of Twitter. The opposite trend thus reduces the performance improvement of Hieroglyph.

Results for Bipart show the superiority of Hieroglyph due to its ability to perform locally-sufficient computation across phase boundaries. Given that each of the four phases in Bipart consists of only one superstep and that messages generated in one phase are always processed by the next phase, Hieroglyph's performance would be identical to that of PowerLyra if local sufficiency is confined within phase boundaries. In other words, system-only support for locally-sufficient computation—without the flexibility introduced in the CSM abstraction—would miss the opportunity of performance enhancement in Bipart. With our CSM bipartite matching design, Hieroglyph achieves up to 3.58x speedup over PowerGraph and 3.49x over PowerLyra.

Comparing against GiraphUC. GiraphUC achieves better performance than both PowerGraph and PowerLyra for all four algorithms on LiveJournal, Wiki, and Road. It, however, becomes inferior on Twitter and Web. GiraphUC's performance variation mirrors that of Hieroglyph. When the computation workload increases in the synchronous execution, the relative effectiveness of local sufficiency reduces.

Yet, speedups of Hieroglyph over GiraphUC for Twitter and Web are larger than those for the remaining datasets. Although the reduced effectiveness of locally-sufficient computation affects both Hieroglyph and GiraphUC in Twitter and Web, Hieroglyph, with its vertex-cut support via CSM, gracefully handles the increasing workload with respect to GiraphUC.

Performance Breakdown. Figure 2.6a shows the performance breakdown of the compute workers, which are responsible for both the compute and the merge stage. For the four algorithms used in our evaluation, the compute stage dominates the workload of the compute

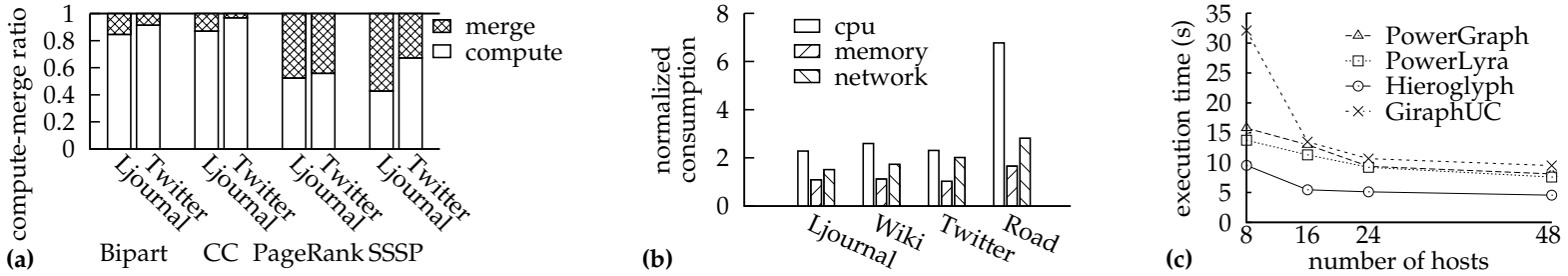

Figure 2.6: Results on performance breakdown, resource consumption, and scalability. (a) Performance breakdown of Hieroglyph's *Compute* and *Merge*. (b) Resource consumption, normalized to PowerLyra. (c) Scalability comparison, with the number of hosts varying from 8 to 48.

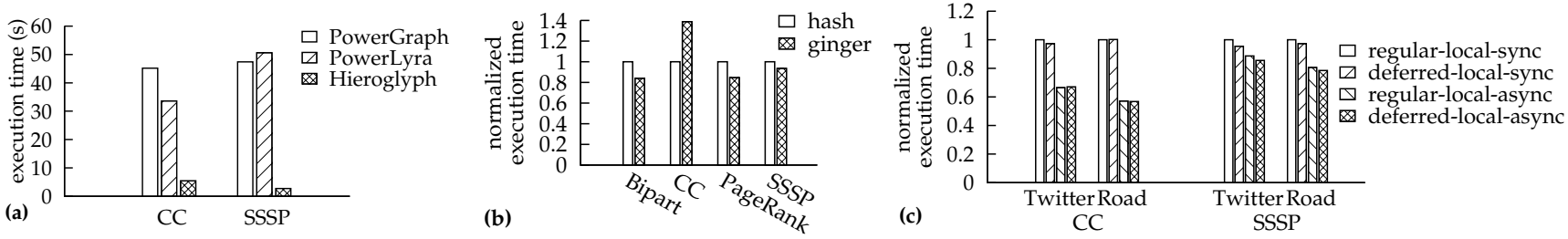

Figure 2.7: Results on asynchronous execution, effects of graph partitioning, and Hieroglyph's optimization using deferred switching and locally-synchronous and asynchronous modes. (a) Performance of PowerGraph and PowerLyra in asynchronous mode. (b) Effect of graph partitioning. (c) Effect of regular vs. deferred switching and locally-synchronous vs. asynchronous execution in Hieroglyph .

workers in most cases. The compute-merge ratio is a function of algorithm and dataset. Figure 2.6a shows that, (*i*) the compute-merge ratio related to the Twitter dataset is higher than that of Livejournal and (*ii*) the ratios related to Bipart and CC are higher than those of PageRank and SSSP.

In general, a larger portion of execution time in the merge stage indicates a higher cost of inconsistency resolution. This cost is determined by two factors: the frequency of the activation of merge stages and the cost of each activation with respect to the cost of compute stages. In the case of SSSP, for example, given that the costs of each activation of the merge stage and the compute stage are comparable, the large portion of execution time in the merge stage indicates frequent activation of the merging logic. Note, however, that both compute and merge stages contribute to the final graph state convergence. A low compute-merge ratio does not entail an insufficient CSM algorithm design.

Resource Consumption. Figure 2.6b shows the resource consumption of Hieroglyph with respect to PowerLyra when executing SSSP. CPU consumption is measured by the number of vertex state updates. Hieroglyph conducts a substantially larger amount of vertex updates, due to the activation of the update function on all replicas of each vertex, instead of only the master replica. Such overhead is also due to the use of potentially inconsistent local state for update in Hieroglyph. Inconsistency resolution incurs a 51%–182% overhead regarding network traffic. Since the communication workers progress independently, however, the negative impact of such an overhead on the overall performance is minimized. Hieroglyph's memory overhead varies from 3% to 65%, thanks to the maintenance of additional states for *Sync* and *Merge*.

Scalability. Figure 2.6c shows the scalability of the four systems when executing CC on Twitter. The execution time of all systems reduces with the increasing number of hosts. Yet, all systems demonstrate sublinear speedup with the increasing number of hosts, due to the intrinsic inter-host dependency of the workload. In terms of execution time, Hieroglyph outperforms the other systems in all our settings. Its speedup varies between 1.66x–3.37x,

Table 2.5: Hieroglyph's speedup over synchronous and asynchronous PowerGraph and PowerLyra

CC	PowerGraph	PowerLyra
synchronous	2.38x	2.07x
asynchronous	8.27x	6.15x
SSSP	PowerGraph	PowerLyra
synchronous	1.94x	1.65x
asynchronous	17.42x	18.58x

2.38x–2.47x, 1.83x–2.08x, and 1.78x–2.08x, when the number of hosts are 8, 16, 24, and 48, respectively.

Asynchronous Execution. When running in the asynchronous mode, the performance of PowerGraph and PowerLyra degrades significantly for CC, PageRank, and SSSP.[12] Figure 2.7a compares the performance of PowerGraph, PowerLyra, and Hieroglyph when executing CC and SSSP on Twitter (summarized in Table 2.5). We choose Twitter for evaluating the asynchronous mode of PowerGraph and PowerLyra, because it produces the minimum speedup for Hieroglyph and thus provides a conservative view of the performance improvement.

For CC, the speedup of Hieroglyph with respect to PowerGraph more than triples when comparing the latter's synchronous execution to asynchronous execution. As for PowerLyra, the speedup of Hieroglyph triples, as well. The increase is even more significant in the case of SSSP. Hieroglyph's speedup boosts from 1.94x for synchronous PowerGraph to 17.42x for asynchronous PowerGraph, enlarging the performance gap by 8x. For PowerLyra, the gap enlarges by over an order-of-magnitude.

Graph Partitioning. Figure 2.7b shows the performance of Hieroglyph using hash and ginger-heuristic graph partitioning [19], measured with the Twitter dataset. Overall, Hieroglyph achieves high performance in both hash and ginger partitioning. Given its ability to perform locally-sufficient computation, Hieroglyph can mitigate the effect of unbalanced

[12] Bipart requires synchronous mode on PowerGraph and PowerLyra.

workload caused by graph partitioning. Yet, in general, Hieroglyph still benefits from more balanced ginger partitioning.

Deferred Switching. The effect of deferring the switching of synchronization state from the beginning of a synchronization iteration to the time when such state is accessed is negligible for CC and SSSP, yielding a maximum of 5% reduction in execution time (cf. Figure 2.7c). Such ineffectiveness may be partly attributed to the high priority assigned to the communication workers in the current implementation of Hieroglyph. While the computation workers frequently yield to the communication workers (e.g., by reducing the vertex chunk size), the latter proceed until all available data have been exchanged. The rationale behind this default mode of Hieroglyph is that, when local state is updated, it is advantageous to propagate the update to all replicas. In other words, it is desirable to minimize the time window during which the vertex state remains inconsistent. Locally-sufficient computation proceeds opportunistically, aiming at making progress to hide communication cost yet minimizing additional communication delay (in the form of reduced responsiveness of the communication workers due to parallel locally-sufficient computation). The probability that the local vertex state is repeatedly updated before synchronization is thus minimized, so is the effect of deferred switching.

There are, nevertheless, cases where it is beneficial to assign high priority to local state propagation [75].[13] In those cases, we expect deferred switching to significantly shorten the execution time.

Local Asynchrony. Figure 2.7c also compares the performance of locally-synchronous execution with that of locally-asynchronous execution. For all cases in Figure 2.7c, local asynchrony leads to superior performance, with the speedup ranging from 1.13x to 1.76x. For both CC and SSSP, the gain of local asynchrony is larger for Road than for Twitter. This is because the effect of fast state propagation in locally-asynchronous mode is amplified by

[13] In graph-centric approaches [66,75], the same problem bares the form of whether to perform per-partition computation iteratively (e.g., until the partition converges) or to frequently propagate updated external vertex state to adjacent partitions.

Table 2.6: Algorithm Complexity in Lines of Code

Algorithm	GAS	CSM
Bipart	123	206
CC	46	88
PageRank	48	99
SSSP	44	88

the large diameter of Road.

It is also worth noting that, for CC and SSSP, vertex-consistency is sufficient for correctness [43]. Since during local-sufficient computation, vertex state can be updated only in the *apply* function, there is no write-write data race. In addition, read-write data race is benign in both CC and SSSP.[14] Consequently, no lock is required for accessing vertex state. Regarding operations on the message queue, message enqueuing requires lock protection in both locally-synchronous and asynchronous modes. The only additional locking overhead induced by locally-asynchronous execution is thus for message dequeuing. This slight overhead is outweighed by the benefit of fast state propagation, leading to the significant improvement of locally-asynchronous execution.

The performance improvement of local asynchrony is encouraging. Yet, locally-synchronous execution has its own merit. For example, it efficiently supports the bipartite matching algorithm, in which active vertices of the current phase send messages to be processed by the subsequent phase. Had Hieroglyph only supported locally-asynchronous mode, it would require the implementation of phase-related message tagging [29], complicating multi-phase algorithm design.

Complexity. Table 2.6 compares the complexity of CSM algorithms with their GAS counterparts, using lines of code as the metric. Using Hieroglyph's implementation of the CSM abstraction, the four algorithms studied in the evaluation require 67%–106% more lines of

[14] For example, assume that a vertex v_i will not send a message to another vertex v_j, if v_i obtains (i.e., reads) the most recent update (i.e., write) of v_j. Then, if v_i sends a message to v_j due to the access of a stale state of v_j but the message arrives after v_j's update, it will be discarded due to program logic, for both CC and SSSP. Such race thus does not affect correctness.

code to be expressed in CSM. Note that, the complexity of algorithm design in CSM is also determined by the inconsistency resolution logic in *Sync* and *Merge*. For CC, PageRank, and SSSP, their corresponding inconsistency resolution logic resembles the logic used in their locally-sufficient computation, the latter an extension of the GAS implementation. As a result, these three algorithms are relatively easy to be implemented. Bipart is more difficult, in contrast, because of the dissimilarity between *Compute* and the inconsistency-fixing logic in *Sync* and *Merge*.

2.5 Related Work

Execution Modes. In this chapter, we have discussed the vertex-centric programming paradigm, as well as synchronous, asynchronous, and barrierless asynchronous execution. We have discussed Pregel [46], PowerGraph [26], PowerLyra [19], and GiraphUC [29] in prior sections of the chapter. Below we summarize other graph processing systems.

Influenced by Pregel, many graph processing systems, such as Giraph [1], Mizan [36], GPS [57], Pregel+ [78], GraM [73], and Quegel [79], follow the bulk synchronous parallel model. Giraph [1] is an open-source implementation of Pregel. Mizan [36] features dynamic workload balancing. GPS [57] broadens the vertex-centric paradigm to support master computation—computation performed by a master host and serialized to BSP supersteps on all hosts. It also introduces dynamic graph repartitioning and large adjacency list partitioning for reducing communication overhead. Pregel+ [78] analyzes the benefit of vertex state mirroring [43] and extends the Pregel abstraction with a request-respond paradigm, enhancing the flexibility in state propagation. GraM [73] achieves overlapping of computation and communication at the architectural level, via a multi-core-aware RDMA-based communication stack. Quegel [79] extends BSP to support superstep-sharing execution, effectively amortizing the cumulative synchronization cost across the parallel execution of multiple queries.

Besides BSP-style systems, PowerGraph [26], Trinity [60], and GRACE [72] support both synchronous and asynchronous modes. PowerSwitch [74] employs Hsync, a hybrid execution mode featuring adaptive switching between synchronous and asynchronous modes for better performance.

Multi-Core/Out-of-Core Processing. Ligra [62] is a high-performance graph processing system targeting a multi-core setting. GraphChi [39] and X-Stream [55] enhances the performance of out-of-core processing by exploiting data access locality. Chaos [54] extends out-of-core processing to a distributed setting. PathGraph [81] uses path-centric computation to improve memory and disk locality. FlashGraph [82] enhances the performance and applicability of out-of-core processing systems by using SSDs. These approaches are orthogonal to Hieroglyph, which targets a distributed environment.

Dataflow Operators. Pregelix [14] and GraphX [27] both map a vertex-centric abstraction onto distributed dataflow operators, enabling efficient support of graph processing over general purpose dataflow engines. Such approaches benefit from the substantial efforts and experiences in the design and implementation of high-performance dataflow engines. They also seamlessly integrate graph processing into a high-level data-analytics framework. Hieroglyph, in contrast, is a specialized graph processing system. Such approach commonly achieves high performance in the graph processing stage—generally exceeding general-purpose approaches—but requires substantial efforts to be integrated with other data-analytics components. Mapping the CSM abstraction onto dataflow operators would enable locally-sufficient computation on graph processing systems built atop dataflow engines, thus improving the latter's performance in the graph processing stage.

Other Programming Abstractions. Pegasus [34] expresses graph algorithms via generalized iterated matrix-vector multiplication that can be efficiently executed on a MapReduce platform. Combinatorial BLAS [15] expresses graph algorithms using linear algebra primitives. Galois [49] implements the amorphous data-parallelism (ADP) model and achieves

high performance due to the high efficiency of a general low-level programming model and sophisticated task scheduling.

Giraph++ [66] advocates the exposure of graph partitions to users, proposing a "think like a graph" paradigm. The graph-centric abstraction enables users to design algorithms from the perspective of all vertices in a partition, leading to more efficient algorithms. Blogel [77] advances the graph-centric abstraction by associating partitions with their own state and promoting them to first-class entities: algorithms can be defined for partitions instead of vertices of a partition. Xie et al. [75] employs a similar block-centric approach in GRACE to the improvement of CPU cache utilization. Chen et al. [20] augments message combination with network-topology-awareness, demonstrating another interesting aspect of the generic hierarchical graph processing idea. P++ [83], GoFFish [63, 64], and GraphHP [17] also apply the graph-centric principle in their system designs. Arabesque [65] proposes a "think-like-an-embedding" paradigm for distributed graph mining. NScale [52] enables users to specify a subgraph where an algorithm should be executed.

Failure Recovery. Most graph processing systems [1, 26, 36] resort to checkpoint-based recovery mechanism. Both vertex state and messages are checkpointed. Similar to its locally-sufficient computation, Hieroglyph performs checkpointing locally, without global synchronization. Only vertex state needs to be checkpointed, due to the nonexistence of inter-host messages. Shen et al. [61] proposes to reassign failed partitions upon failure in order to balance the computation and communication cost related to failure recovery among computing hosts. Outgoing messages are logged per superstep so that vertices outside the failed partitions do not need to be recomputed: replaying logged messages is sufficient for recomputing failed vertices. Zorro [51] employs a reactive approach, trading accuracy in recovered execution for zero overhead during failure-free execution.

Performance Evaluation. With the proliferation of large-scale graph processing systems comes the need for thorough performance evaluation. Existing efforts [28, 30, 44, 59] aim at providing objective comparisons regarding system performance and resource consumption,

facilitating users to make educated selection among candidate processing systems.

2.6 Conclusions

In this chapter, we introduced the Compute-Sync-Merge (CSM) abstraction, a vertex-centric programming paradigm enabling locally-sufficient computation over vertex-cut partitioning. We demonstrated the expressiveness of CSM by implementing several fundamental algorithms with it. We showcased CSM's potential in enabling highly-efficient multi-phase algorithm design with our bipartite matching implementation, where locally-sufficient computation freely proceeds across phase boundaries and inconsistent local state is fixed in CSM's *Sync* and *Merge* stages. Hieroglyph—our prototype system supporting the CSM abstraction—outperforms state of the art by up to 53x.

CHAPTER III

Multi-Version Graph Processing

In the second half of this thesis, we focus on a specific scenario in large-scale graph processing: multi-version graph processing. This chapter starts with a discussion on the motivation for multi-version graph processing, presenting several concrete and compelling use cases. It then presents the multi-version graph processing terminology used in this thesis, followed by a discussion on how a multi-version graph evolves. This chapter concludes with how a multi-version graph processing task is conducted in state-of-the-art systems.

3.1 Motivation

Large-scale real-world graphs evolve in various ways [67, 84]. Conceptually, if we define a certain state of an evolving graph to be the *root version*—that is, the common ancestor—of the graph and consider all the other graphs derived from it as its successor versions, then the collection of graphs becomes a multi-version graph.

Accurately capturing and understanding the dynamics of graph evolution increase the usefulness of graph processing: there exists information that can be extracted only by processing and comparing multiple versions of a graph. For example, Ren et al. [53] note that the varying closeness of the friendship between two users u_i and u_j in a social network can be studied by processing multiple versions of the social network graph, each corresponding to a temporal snapshot of that network. Fine-grained insight can be drawn from

such analysis, for example, about whether the increasing closeness of u_i and u_j is due to the appearance of a new common friend or the establishment of friendship between two previously non-adjacent users u_k and u_l along the shortest path between u_i and u_j.

Another case of multi-version graph processing is collaborative graph editing and analysis from multiple data scientists [8]. Starting from a shared common graph dataset, each data scientist may need to perform a different type of analysis, thus advancing the graph to a specific successor version. Such versions may later be combined to construct another shared version or used as input for cross-version comparison.

Multi-version graph processing becomes even more important and prevalent, if we assume an *as-a-service* perspective common in today's big data analytics. When graph processing is offered as a service, individual graph processing requests from independent users can become related from the perspective of a service provider. This is the case when requests from independent users each targets one of the many versions of the same multi-version graph. For example, suppose several users send requests to a navigation service provider, requesting the shortest route from Location A to Location B. Suppose that each user has some specific constraint, for example, avoiding a certain portion of a highway or requesting Location C as a via point. Also suppose that the service provider solves the navigation problem with a shortest-path algorithm. Then in this scenario, the service provider needs to perform a multi-version graph processing task, running the same algorithm on multiple versions of the same road network graph, each version representing the specific constraint of a user.

3.2 Terminology

We define terminology used for discussing multi-version graph processing in this thesis.

Version. A version corresponds to specific state of an evolving multi-version graph. Both graph topology and properties—in the form of vertex and edge state—are associated with

a version.

All versions are not of the same importance. For example, if a user creates a successor version v_b by removing, from version v_a, edges satisfying a certain criteria. Suppose n edges in v_a need to be removed. Then removing one edge at a time would lead to $n-1$ intermediate versions, which do not have specific meaning for the user, do not have version numbers or identifiers, and are thus not externally addressable. We call such intermediate versions *internal versions* and v_a and v_b *external versions*. In this thesis, we use the term *versions* to refer to external versions.

When the state of a vertex related to a version is updated during computation, it does *not* lead to a new version of the vertex. Instead, such computation-related update preserves the version information. For instance, when the rank of a vertex in a version is updated in an iteration of the PageRank algorithm, it is still associated with the same version. Upon completion of the computation, users decide whether to use the updated state to overwrite the existing state of the same version in persistent storage or to label the updated state with a new version number.

We assume the existence of a common subgraph among all versions, whose size is substantially larger than the difference between two versions. This is the case, for example, for temporal snapshots of large social networks in which links identical across all snapshots significantly outnumber newly established links in a fixed relatively-short time window [48, 53, 71].

Graph. A graph is a collection of graph versions. For simplicity, we assume that there exists a common ancestor for all versions of a graph.

Root version. The common ancestor version of a multi-version graph is called the root version, or the default version. All other versions are referred to as non-root versions.

Delta. A delta represents the difference between two versions. For example, if version v_b can be created by adding an edge e to another version v_a, then the delta δ_{ab} connecting v_a

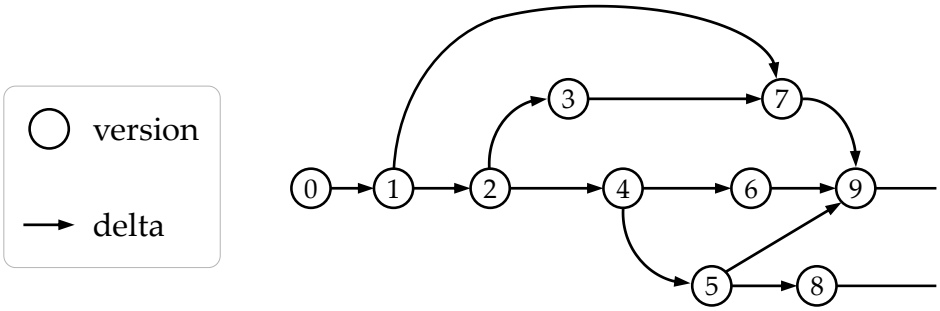

Figure 3.1: Generic graph evolution

and v_b consists of e.

The root version of a graph can evolve with the graph. When the root version evolves, other versions represented as deltas atop the root version need to be adjusted.

3.3 Graph Evolution

Graphs may evolve linearly or non-linearly. Most existing work [37, 53] adopt a linear evolution model. This model is sufficient for expressing temporal graph snapshots—a prominent scenario in multi-version graph processing.

This thesis intends to study multi-version graph processing in a broader context, as showcased in Section 3.1. As a result, we adopt a generic graph evolution model (cf. Figure 3.1).[1] Versions can branch off a common ancestor version or merge into a common successor version.

3.4 Style of Multi-Version Graph Processing

Multi-version graph processing can proceed sequentially or in parallel. We call the former *sequential multi-version graph processing* and the latter *parallel multi-version graph processing*.

[1] Our generic graph evolution model is conceptually similar to the generic dataset evolution model proposed in DataHub [8].

Most state-of-the-art graph processing systems target standalone graph processing tasks. They are thus unaware of versions. Each version of a multi-version graph is processed independently. Assuming abundant computing resource and infinitely fine-grained billing in a cloud environment [4], there is little distinction between sequential and parallel multi-version graph processing from the perspective of a version-unaware graph processing system: resource and time become interchangeable and the total cost remains the same. For example, if processing one version requires r computing resource, finishes in t, and incurs a cost of c, then processing n versions incur a cost of nc, regardless of whether the n versions are processed sequentially, using r resource and finishing in nt, or in parallel, using nr resource and finishing in t.

In this thesis, we show the deficiency of ignoring the relation among versions in multi-version graph processing. Regarding augmenting graph processing systems with version awareness, sequential and parallel processing need to be addressed from different perspectives, due to their distinct characteristics in graph fetching and computation stages.

CHAPTER IV

Version Traveler: Fast and Memory-Efficient Version Switching in Sequential Multi-Version Graph Processing

In this chapter, we first study the design space and optimization opportunities in sequential multi-version graph processing systems. We then introduce our solution—Version Traveler, a sequential multi-version graph processing system preparing in-memory graph representation with cached graph state—in Section 4.2. We evaluate its efficacy in Section 4.3. Section 4.4 surveys additional related work. Section 4.5 concludes the chapter.

4.1 Multi-Version Graph Processing

In this section, we first discuss the characteristics of multi-version graph processing workloads, followed by a discussion on its workflow. We then summarize related work, analyze the design space for efficient multi-version graph processing systems, and discuss challenges.

4.1.1 Workload Characteristics

In multi-version graph processing, version switching commonly demonstrates randomness and locality. Version switching is *arbitrary*, in that the next version may precede or

succeed the current version in the graph evolution.[1] Such a switching sequence may be dynamic, unable to be predetermined by the graph processing system. Version switching is *local*, in that the next version commonly resides within the vicinity—in terms of similarity—of the current one in the graph evolution.

We exemplify the demand for arbitrary local version switching with three examples. First, suppose we need to identify the cause of the varying distance between two users in a social network [53]. For simplicity, assume that the distances in versions i and k are different and that the distance changes only once along the evolution from versions i to k. If, after processing version $j = \frac{i+k}{2}$, a binary-search-style exploration algorithm finds that the distance in that version remains the same as that in version k but differs from that in version i, then the algorithm would invoke another iteration of shortest distance computation for version $m = \frac{i+j}{2}$. The switching from versions j to m is arbitrary for the supporting graph processing system. In terms of locality, although the search may oscillate between versions with high dissimilarity at the beginning, the version locality increases exponentially with the progress of the execution.

Second, in interactive big data analytics, an analyst may rerun an algorithm on a graph version, after digesting the results of the previous execution. Which version should be processed next depends on the analyst's understanding of the existing results, as well as his/her domain knowledge and intuition. This leads to arbitrary version switching from the perspective of the graph processing system. As for locality, such analysis commonly follows a refinement procedure, where significant efforts are required to zoom in and conduct in-depth analysis on a cluster of versions within the vicinity of each other.

Third, in a collaborative data analytics environment, both datasets and computing power are shared among users [8]. Individual tasks—each targeting a graph version—can be combined by the processing system, leading to multi-version graph processing. Since the next request may be enqueued during the processing of the current version and may target a ver-

[1] More broadly, in a non-linear graph evolution scenario [8], the next version may reside in a different branch than the current version.

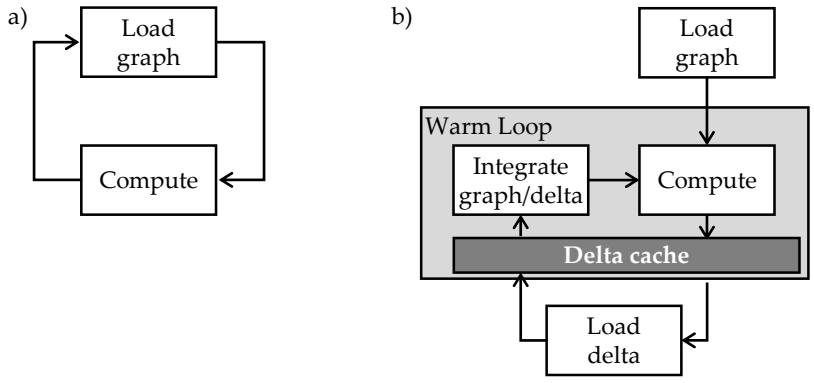

Figure 4.1: Version switching workflow, (a) with and (b) without the use of deltas

sion preceding or succeeding the current version in the graph evolution, version switching is arbitrary. Regarding locality, independently-submitted tasks may target similar versions. Such is the case, for example, where various algorithms are employed to capture and understand trending events in an evolving social network.

4.1.2 Workflow

A typical multi-version graph processing workflow is divided into multiple iterations. In each iteration, an arbitrary graph version is processed. Systems designed for individual graph processing tasks are unable to recognize or take advantage of the evolution relation among versions. Treating each version as a standalone graph, such systems first fully load the version from persistent storage into memory and then execute a user-defined graph algorithm over it (cf. Figure 4.1(a)).

When versions of a working set share a substantial common subgraph, working with deltas—representations of the differences between graph versions—can be more efficient. Figure 4.1(b) shows the multi-version processing workflow with deltas. After the first version is loaded and processed, switching to a subsequent version can be achieved by integrating the current version in memory with deltas relating the current and the next versions. In general, deltas are much smaller than full graphs [53, 71]. As a result, they can be cached in memory, further improving the efficiency of version switching.

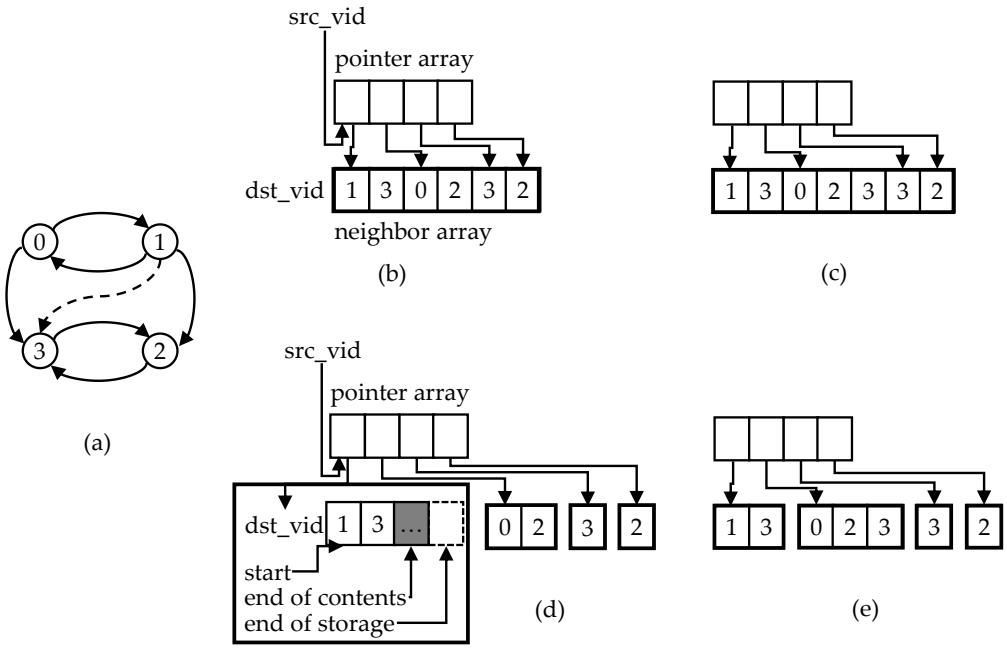

Figure 4.2: Graph representations: (a) illustrates two versions of a graph. A circle represents a vertex (vertex id inside) and an arrow represents an edge (edge id omitted). The first version consists of solid-arrow edges. The second version has one more edge (illustrated by a dashed arrow). (b) and (c) demonstrate the CSR representation of the out-edges of the two versions. (d) and (e) demonstrate the vector of vectors format. For clarity, each element in the neighbor array in (b)–(e) shows only the destination vertex id and omits the edge id.

4.1.3 Related Work on Graph/Delta Designs

We focus the discussion of related work on in-memory graph and delta representations, because they determine the efficiency of the switching loop (cf. Figure 4.1(b)).[2] For graphs, we specifically focus on representations related to neighbors of vertices, because they differentiate graphs from regular table-form datasets.

We study related work by asking the following three questions:

- Does it provide high computation performance? In particular, does it support fast access of the neighbors of a vertex?[3]

[2]Both graphs and deltas may have different representations in memory and on disk. We focus on in-memory representations, due to their significance in the warm loop of the version switching workflow.

[3]In this chapter, we equate computation performance with neighbor access efficiency for two reasons. First, computation related to graph algorithms affects all systems in the same way and is out of scope. Second,

- Does it support fast version switching?
- Does it store graphs and deltas compactly?

Graph. We study two common graph representations: compressed sparse row (CSR), adopted in PowerGraph [26] and GraphX [27], and a vector-of-vectors (VoV) design, adopted in Giraph [1].

In CSR (cf. Figures 4.2b and 4.2c), all neighbors are packed in an array. A pointer array maintains the address of the first neighbor of each vertex. The set of neighbors for vertex i is thus marked by the values of vertices i and $i+1$ in the pointer array. This representation enables fast access to a vertex's neighbors. Version switching is slow, however. This is because modifying a vertex's neighbor affects pointers and neighbors of all vertices following the one being modified.

As for VoV (cf. Figures 4.2d and 4.2e), the first-level vector functions as the pointer array in CSR, locating the neighbors of a vertex according to the vertex id. Each second-level vector represents the neighbors of a vertex. This format also supports fast neighbor access. In addition, the neighbors of a vertex can be modified without affecting other vertices, thus enabling fast version switching. Its shortcoming is the memory overhead due to maintaining auxiliary information, such as the start and end positions of each vertex's neighbors.[4]

Delta. Previous work has used a compact log-format structure to represent deltas in streaming processing [21]. A log delta consists of an array of log entries, each specifying an edge via its source and destination vertex ids (and an optional edge id) and whether the edge should be added or removed (i.e., an opcode). Log deltas are compact and have no negative impact on the neighbor access efficiency during graph processing. This is because

in the computation stage, a system supports neighbor access and vertex/edge data access. Assuming the storage of data in sequence containers and their identical impact on all systems, computation performance is determined only by neighbor access efficiency.

[4]Such overhead is non-trivial. For example, a 24-byte per-vector overhead (cf. Figure 4.2(d), "start," "end of contents," and "end of storage" pointers each consume 8 bytes) amounts to a 40% overhead for representing the entire out-neighbor array for the Amazon dataset [11, 12], assuming 4-byte vertex/edge ids.

log deltas are conceptually applicable to all graph representations as-is. During graph-delta integration, log deltas are fully absorbed in the graph version. The cost of graph-delta integration is high, however, because all log entries in deltas relating the current and the next versions need to be applied during a version switching.

Alternatively, a system could co-design graph and delta representations to minimize the integration cost. For example, affected neighbor vectors of a VoV graph may be copied and updated in a delta, reducing version switching to simple and fast vector pointer updates but losing compactness. LLAMA [45] partially mitigates the compactness loss by separating modified neighbors related to a version into a dedicated consecutive area in the neighbor array, avoiding copying unmodified neighbors. CSR's pointer array is transformed to a two-level translation table. The first level consists of per-version indirection tables, each bookkeeping a set of second-level pages associated with a version. A second-level page contains a series of vertex records—equivalent to a fragment of CSR's pointer array—with each record indicating the start of a vertex's neighbors.[5] LLAMA's version switching incurs nearly zero time cost: only an indirection table pointer needs to be updated. Its use of page-level copy-on-write for the second-level pages holding vertex records, nevertheless, requires the copy of an entire page even if only one vertex in the page has a modified neighborhood, hindering its compactness.

GraphPool [37] maintains the union of edges across all versions in the graph. Its deltas are per-version bitmaps over the graph's edge array, where a bitmap's n-th bit indicates the existence of the corresponding edge in that version. Version switching is simple: a bitmap pointer is adjusted to point to the next version. In the computation stage, however, this approach requires bitmap checking for determining whether an edge exists in the current version, incurring neighbor access penalty.

[5]Neighbors belonging to the same vertex but stored in separate areas—each containing per-version modifications—are concatenated via *continuation records* such that only one start position needs to be maintained for a vertex's neighbors per version.

4.1.4 Design Dimensions and Challenges

Summarizing lessons learned from related work, we point out that the design of graph and delta must balance between three dimensions: extensibility, compactness, and neighbor access efficiency.

Extensibility. Efficient version switching requires that a delta be easily integrated with a graph. From a data structure perspective, it requires that the neighbors of a vertex be easily extended to reflect the evolution from one version to another. This, in turn, requires that either the data structure representing the neighbors of a vertex support efficient modification (i.e., insertion and removal) or the collection of the neighbors of a version be easily replaced by that of another version.

Compactness. Compact graph and delta representations enable caching a large number of versions, leading to low delta cache miss rate and high version switching efficiency. Moreover, real-world large graphs commonly have millions of vertices and millions or billions of edges, making the compactness of the neighborhood data structure a primary requirement.

Access Efficiency. A common and crucial operation during computation is to access a complete collection of neighbors for a vertex. Fast neighbor access requires limiting the number of lookups in the integrated graph/delta data structure. Ideally, only one lookup is sufficient for locating the first neighbor of a vertex. The remaining neighbors can then be accessed sequentially.

The main design challenge is to carefully balance the requirements from the above three dimensions and co-design graph and delta representations such that they are extensible, compact, and efficient in neighbor access. Achieving the balance is difficult, as witnessed by existing designs, because those requirements commonly lead to contradicting design choices.

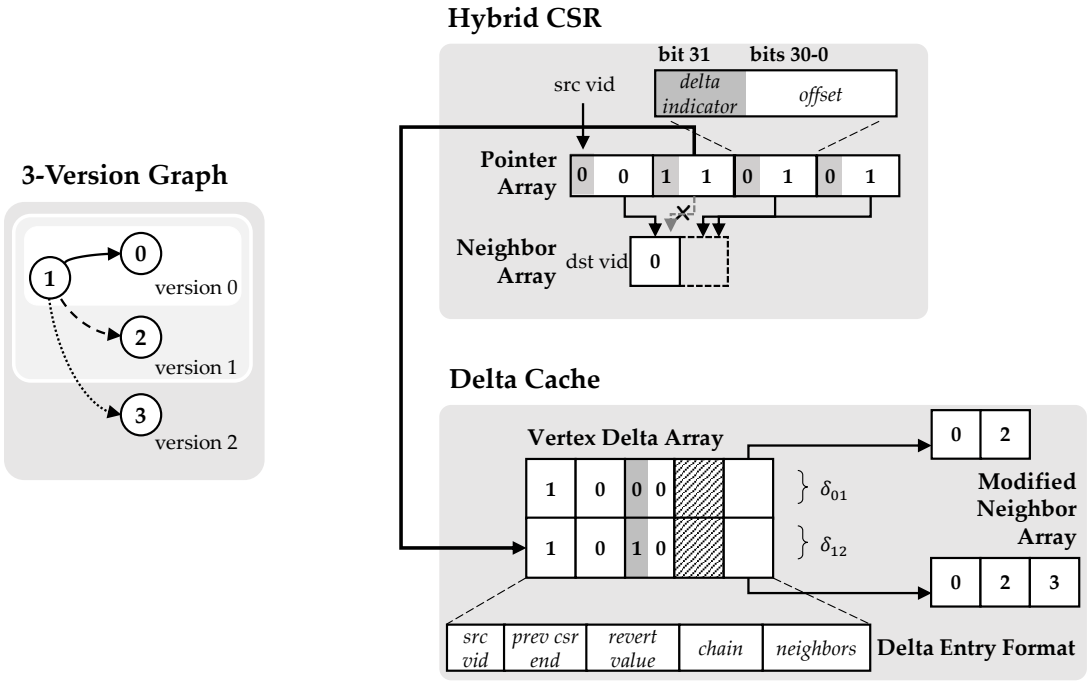

Figure 4.3: Hybrid graph representation

4.2 Version Traveler

We introduce Version Traveler (VT), a graph processing system that features a graph/delta co-design achieving compactness, extensibility, and access efficiency. Residing in the core of VT are two innovative components—a hybrid graph and a hybrid delta cache—bringing together fast neighbor access and compactness of CSR and high extensibility of VoV.

4.2.1 Hybrid Graph

VT's hybrid graph augments CSR in a way that achieves extensibility while remaining compact and efficient in neighbor access (cf. Figure 4.3). It avoids costly in-place modification to CSR's neighbor array by storing vertices with a modified neighborhood in a version delta cache.

Algorithm 8 Neighbor Access

```
1: input vid
2: if csr_ptrs[vid].in_delta = true then
3:     return cache[csr_ptrs[vid]].nbrs
4: else
5:     if csr_ptrs[vid + 1].in_delta = false then
6:         return csr_nbrs[csr_ptrs[vid], csr_ptrs[vid + 1]]
7:     else
8:         return csr_nbrs[csr_ptrs[vid], cache[csr_ptrs[vid + 1]].prev_csr_end]
```

Algorithm 9 Delta Application

```
1: input δ_ij, opcode
2: for e in δ_ij do
3:     if opcode = apply then
4:         csr_ptrs[e.src_vid] ← offset(e)
5:         csr_ptrs[e.src_vid].in_delta ← true
6:      else csr_ptrs[e.src_vid] ← e.revert_value
```

CSR's neighbor array is created during the loading of the first version—also referred to as the root version—and then remains constant. Each subsequent version is loaded into the delta cache, in the form of a series of *vertex delta entries*. Each entry contains information related to the updated neighbors of a vertex, as well as metadata to support neighbor access and version switching (cf. Figure 4.3). VT reserves a *delta indicator bit* in each entry of CSR's pointer array to indicate the placement of a vertex's neighbors for the current version: in CSR's neighbor array or in the vertex delta cache.

Neighbor Access. In a conventional CSR, neighbors of vertex *vid* are bounded by the pointers of vertices *vid* and $vid + 1$. For VT's hybrid CSR, neighbor access may be directed to either CSR's neighbor array or the *neighbors* field of a delta entry, depending on whether the neighbors are stored (cf. Algorithm 8). Each delta entry maintains the end position of the preceding vertex's neighbors in CSR's neighbor array (in the *prev_csr_end* field), such that the end position of *vid* can be determined regardless of whether neighbors of $vid + 1$ are stored in CSR's neighbor array or the delta cache (cf. lines 5–8).

Delta Application and Reversion. VT performs version switching by applying or revert-

ing deltas (cf. Algorithm 9). When applying δ_{ij} to switch from versions i to j, VT iterates over entries belonging to δ_{ij} in the delta cache and, for each entry, updates the corresponding entry (according to the *src_vid* field) in the CSR pointer array with the delta entry's offset in the delta array and sets the delta indicator bit. Reverting δ_{ij} consists of restoring the *revert value* field—which contains the saved value for version i's CSR pointer entry—to the corresponding entry in the CSR pointer array for each entry in δ_{ij}.

Example. We use a 3-version graph in Figure 4.3 to illustrate neighbor access and version switching. The hybrid CSR in Figure 4.3 represents the state of version 2. The three out-neighbors of vertex 1 can be accessed from its delta entry (δ_{12}). For vertex 0, neighbor access requires obtaining the start position from its CSR pointer, due to its cleared delta indicator bit, and the end position from *prev_csr_end* of vertex 1's delta entry. The difference between the two is 0 ($0-0=0$), indicating that vertex 0 has no out-neighbors. In order to switch from versions 2 back to 1, VT reverts δ_{12}, which has only one entry related to vertex 1. Its *revert value* field, of which the delta indicator bit is set and the offset is 0, is restored to vertex 1's CSR pointer entry. After reversion, vertex 1's CSR pointer entry will point to the first entry in the delta array, which corresponds to vertex 1's delta entry for version 1.

4.2.2 Hybrid Delta

For simplicity, in Section 4.2.1, we assume that a delta entry maintains the entire neighbors of a vertex (cf. Figure 4.3). This is memory-inefficient for vertices with a large number of neighbors and small amount of per-version modifications, due to numerous redundant copies of neighbors. To improve compactness, we propose two complementary solutions: *Sharing* and *Chaining*. Sharing preserves access efficiency and trades extensibility for compactness. Chaining preserves extensibility and trades access efficiency for compactness.

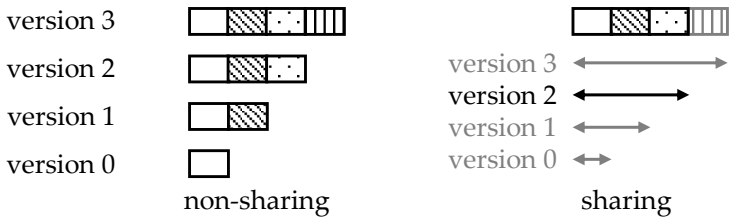

Figure 4.4: Illustration of the concept of Sharing

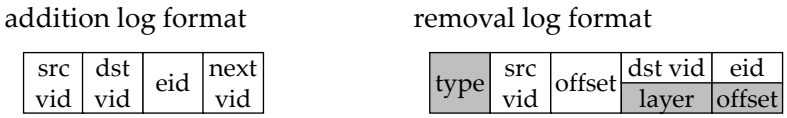

Figure 4.5: Delta log format

4.2.2.1 Sharing

Concept. Sharing reduces the memory footprint by merging a vertex's delta entries spanning multiple versions into one shared entry. Figure 4.4 shows an example with four versions of a vertex, each adding one neighbor to its base. When they share a delta entry, there exists only one neighbor vector, containing the neighbors of the vertex related to the current version being processed. The challenge is to compactly specify how the shared vector is modified during version switching. VT maintains this information in addition and removal *delta logs*.

Delta Representation. In the Sharing mode, VT does not create delta cache entries with copies of modified neighbor arrays. Instead, it creates log entries: specifying the neighbors it would have added to or removed from the neighbor array in an addition or removal log. Each entry in the addition log array (cf. Figure 4.5) contains the source and target vids of the added edge, as well as the edge id. Logs associated with the same vertex (the source vid in the out-neighbor case) are continuously stored. A *next vid* field indicates the start position of the logs associated with a subsequent vertex. The format of a removal log entry—gray fields apart—is similar. Its *offset* field refers to the offset within the neighbor array where the removal should take place.

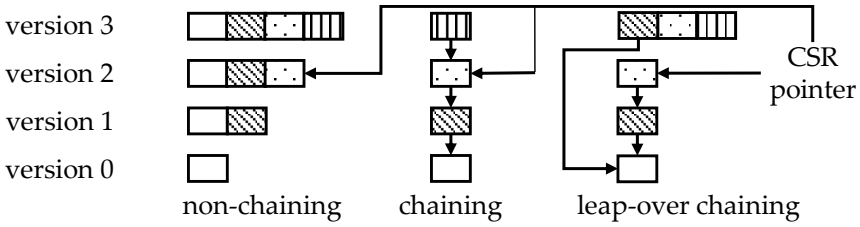

Figure 4.6: Illustration of the concept of Chaining

Neighbor Access. Sharing has no effect on neighbor access. When multiple versions of a vertex share a neighbor vector, the CSR pointer header points to the same delta entry containing the vector for all versions.

Delta Application/Reversion. In the Sharing mode, version switching resorts to log-based delta application/reversion, similar to streaming processing [21, 39]. During delta application, for an neighbor addition, the neighbor is simply appended to the end of the neighbor vector. For a removal, VT removes the neighbor at the offset according to the *offset* value in the removal log entry. Delta reversion follows the inverse procedures.

4.2.2.2 Chaining

Concept. Chaining refers to the representation of a vertex's neighbors with a chain of vectors, each containing a subset of neighbors and capturing the difference between the version associated with it and its base version. In Figure 4.6, with each version chained onto its base, only one neighbor needs to be maintained per version. Redundant copies are eliminated, improving compactness. Extensibility remains the same: to switch from versions 1 to 2, for example, we need to adjust only the CSR pointer to version 2's delta entry, regardless of whether that entry's neighbor vector is chained onto another. Access efficiency decreases in Chaining, because of the need to switch among multiple neighbor vectors. Chaining imposes two new challenges to delta design: chaining beyond the base, called *Leap-Over Chaining*, and removal from ancestors, called *indirect removal*.

Delta Representation. Leap-Over Chaining intends to accelerate neighbor access. In

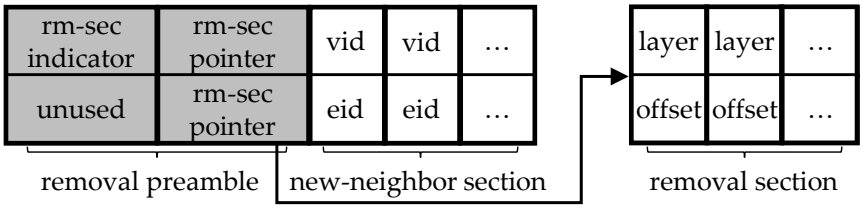

Figure 4.7: Neighbor vector format

Figure 4.6, with each version chained onto its base, the neighbor access for later versions leads to considerable performance hit, limiting Chaining to a small set of adjacent versions in the graph evolution relation. Leap-Over Chaining enables the chaining of a delta entry on an indirect ancestor version. For instance, version 3 in Figure 4.6 can be chained onto version 0.

To support Chaining, in particular Leap-Over Chaining, we introduce a *chaining* field to the delta entry format (cf. Figure 4.3). When Chaining is disabled,[6] the entry is a standalone entry with a complete copy of neighbors. When Chaining is in use, VT saves a pointer to the entry upon which the current one is based along the chain to the latter's *chaining* field. Similar to CSR pointers, a chaining pointer uses its most significant bit to indicate whether the offset is for CSR or for the delta array.

To support indirect removal, a neighbor vector is divided into two sections: a new-neighbor section and a removal section (cf. Figure 4.7). An element in the new-neighbor section represents a new neighbor added to the vertex in the current version. An element in the removal section corresponds to a removed element, with *layer* indicating the neighbor vector in the chain where the removal should take place and *offset* the position of the to-be-removed neighbor within the vector. To improve compactness, we overload the first element in the new-neighbor section: it is marked with a special flag if the removal section exists, in which case the second element contains a pointer to the removal section;[7] otherwise it contains the first added neighbor.

[6] That is, setting all bits in *chaining* to one.

[7] The first two elements are also referred to as *removal preamble*.

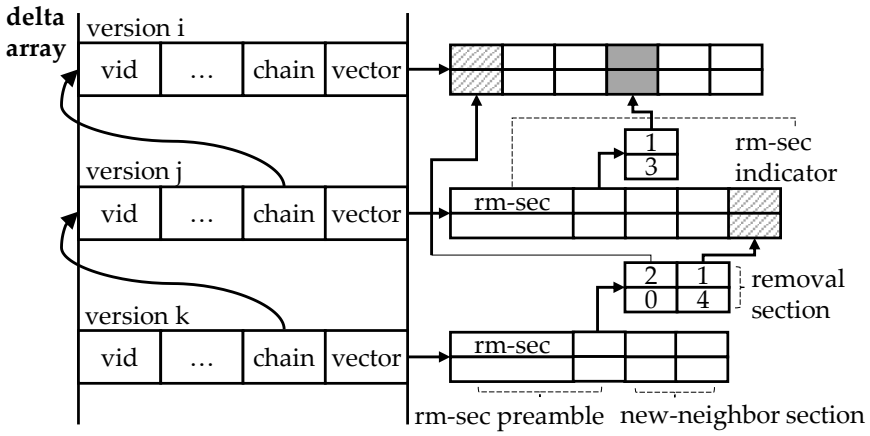

Figure 4.8: Chaining in delta array entries

Example. We follow a concrete example illustrated in Figure 4.8 to present in detail VT's support for neighbor removal in the Chaining mode. In that figure, there are three entries in the delta array associated with a vertex in three versions i, j, and k (from top to bottom). The top entry is a standalone entry, with a complete copy of the vertex's neighbors related to version i. The middle entry has its chaining pointer set to the top one, indicating its dependency on the latter. The removal section indicator of its neighbor vector is turned on. As a result, the next element following the removal section indicator contains a vector pointer, holding the address of the removal section associated with that delta entry. The removal section contains only one element. Its *layer* field is set to 1, indicating that the neighbor vector from which the neighbor should be removed is one step away from the current one, that is, the vector of version i. Its *offset* field is set to 3, indicating that the gray element in version i, whose offset is 3, should be removed when neighbors in version j are accessed. Similarly, the neighbor vector of version k has a removal section and a new-neighbor section. The two elements in its removal section point to the 0th and the 4th elements in versions i and j, respectively.

Effect on Removal Log. Since Chaining introduces the separation of new-neighbor and removal sections, Sharing's removal log format needs to be adjusted (cf. gray fields in Figure 4.5). Specifically, a *type* field is added to differentiate the two sections. A {layer, offset}

pair in a removal section is stored similarly to a {vid, eid} pair in a new-neighbor section. During delta application, if a removal takes place in the current vertex's new-neighbor section, then the corresponding {vid, eid} pair is removed. Otherwise, the {layer, offset} pair is inserted to the current vertex's removal section. The inverse procedure achieves delta reversion.

Neighbor Access. Given a vertex, VT first locates its entry via the CSR pointer header. If the neighbors are stored in a delta array entry whose Chaining mode is turned on, then VT iteratively accesses neighbors stored in entries along the chain, skipping neighbors that no longer exist in the current version using the removal sections. Otherwise, it follows the common neighbor access procedure, as described in Section 4.2.1.

Delta Application/Reversion. Except for Chaining's aforementioned effect on removal logs, both delta application and reversion follow the description in Section 4.2.1.

4.2.2.3 Relationship between Chaining and Sharing

Chaining and Sharing are similar in that they both aim at reducing memory consumption by storing only the difference among versions. Sharing is a good choice when the size of neighbors is large and the delta size is small. A large neighborhood leads to a considerable gain in compactness over a full-neighbor-copy approach, whereas a small delta entails a moderate cost for log-based version switching. Chaining is useful when both the sizes of neighbors and delta are large. Similar to Sharing, a large neighbor size leads to a substantial gain in compactness for Chaining. A large delta entails Chaining's superiority to Sharing, due to the avoidance of the latter's costly log-based version switching procedure.

Another way to compare the two is when the concatenation of neighbors occurs. Chaining performs the concatenation in a chain at the computation stage. Sharing performs the concatenation at the version switching stage. Due to the different delta formats used in Chaining and Sharing, the concatenation in Chaining is lighter-weight than that in Sharing. As for the number of concatenation performed for a vertex, the concatenation in Chaining

needs to take place when a vertex's neighbors are accessed. The cost of concatenation in Chaining is thus magnified if a vertex is iteratively processed by an algorithm. The concatenation in Sharing is, in contrast, guaranteed to be once per vertex per version switching.

VT supports Sharing and Chaining as operation modes, complementing the default full-neighbor-copy mode (referred as Full mode). It enables them when the estimated cost of version switching and the potential impact on the computation stage are justified by the amount of memory saving. The current VT implementation supports flexible threshold-based policies: when creating a new delta for a vertex, VT feeds the number of neighbors in its base version and the current delta size related to that vertex to a configurable policy arbitrator function, which determines the activation of Sharing or Chaining.

4.2.3 Implementation

We implement VT by integrating it with PowerGraph [26], replacing the latter's graph representation with VT's hybrid CSR graph and delta/log arrays. VT operates seamlessly with PowerGraph's computation engine layer, thanks to its full support of the same computation-stage graph abstraction viewed from an engine. This also demonstrates VT's broad applicability to existing graph processing systems.

4.3 Evaluation

We first demonstrate the three-way tradeoff among extensibility, compactness, and access efficiency, showing the relative advantage of Full, Chaining, and Sharing. We then compare the performance of VT against PowerGraph and several multi-version reference designs.

4.3.1 Microbenchmark

Design. The goal of microbenchmarking is to evaluate the relative effectiveness of Full, Sharing, and Chaining in balancing the three-way tradeoff. Since the overall tradeoff on a graph is the accumulative effect of the same tradeoff on each vertex, we conduct microbenchmarking from a vertex's perspective. Trends in the microbenchmark results are applicable to varying graph sizes, given the accumulative nature of the per-vertex tradeoff.

We construct a graph with 1000 identical high-degree stars. For each star, only the *center vertex* has a non-empty set of out-neighbors—default to 1000. Each center vertex thus provides the opportunity for an in-depth study of the per-vertex tradeoff. To evaluate it in a multi-version scenario, we create two versions: a source version and a target version. The target version differs from the source by randomly adding or removing out-neighbors of center vertices.

Two key factors related to a delta are its size and the ratio of additions to removals. Prior work shows that the difference between consecutive versions is commonly within 1% of the graph size [53]. For each star, given the default 1000 edges in the base version, we vary the delta size from 1 to 100, corresponding to 0.1% to 10% of the size of the star. The total vertices and edges in the graph thus vary between 0.9 to 1.1 million. We also fix the operation types in a delta: a delta consists of either edge additions or edge removals.

We evaluate extensibility by measuring the version switching time from the source version to the target version, neighbor access efficiency by measuring the time for iterating through all the out-neighbors of center vertices in the target version, and compactness by measuring the memory used for maintaining the graph connectivity information of both versions. All measurements are conducted on a host with 8 3GHz vCPUs and 60GB memory.

Version Switching. Figures 4.9a and 4.9b compare the version switching performance. The performance of Full and Chaining is comparable and remains constant, regardless of

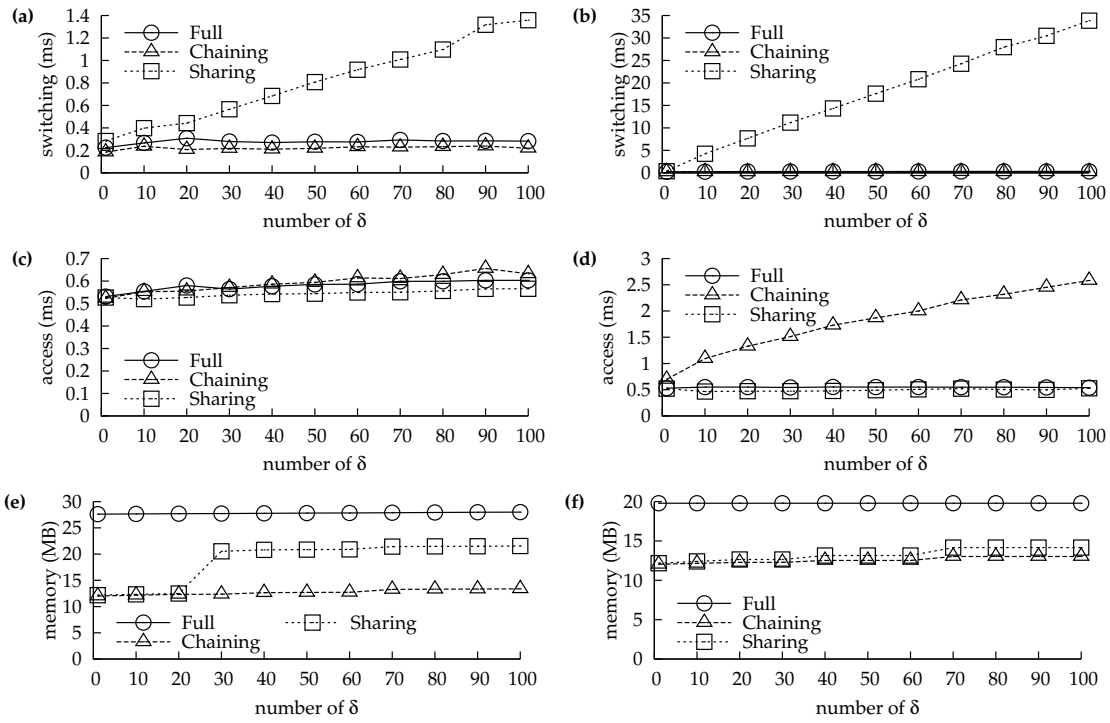

Figure 4.9: Microbenchmark results

the edge modification types in deltas or the delta size, because both approaches require adjusting only CSR pointer values for center vertices. The cost of Sharing linearly grows with the delta size, due to the need to parse a log array whose size is proportional to the delta size. Comparing edge additions with removals, the cost of the former is significantly lower than the latter. This is because, additions translate to appending neighbor records to the end of the neighbor vector and removals involve data movement within the vector.

Access. Figures 4.9c and 4.9d compare the neighbor access speed. Full and Sharing perform equally well for both additions and removals. Since the cost of neighbor access is proportional to the neighborhood size, it linearly increases and decreases with the size of delta in the cases of addition and removal, respectively. Chaining leads to the worst performance in both cases, due to its cost of indirection. The cost is moderate in the case of edge additions, because there is one and only one indirection during neighbor access—that is, the switching from the newly added neighbors to the existing ones—regardless of the delta size. The cost of indirection becomes significant for edge removals, because each

removal separates a previously continuous neighbor range into two, introducing one more indirection during neighbor access. The cost thus linearly grows with the delta size.

Memory. Figures 4.9e and 4.9f show the memory footprint. In both addition and removal cases, Chaining and Sharing lead to significant memory savings comparing with Full. Intuitively, the cost of Full linearly grows with the delta size in the addition case and linearly decreases in the removal case. Our measurements, however, show mostly constant memory footprints in both cases, due to (1) the capacity doubling effect and (2) no capacity reduction upon removal in the vector implementation in our testbed (glibc 2.15). For Chaining and Sharing, the memory footprint grows with the size of delta, regardless of the type of edge modifications. This is because, for both additions and removals, Chaining needs to maintain the modifications either in the neighbor vector (for additions) or in the removal section (for removals). Similarly, Sharing maintains the modifications in the log arrays.

4.3.2 Macrobenchmark

Reference Designs. We compare VT with PowerGraph [26]—a high-performance system targeting individual graph processing—and four reference multi-version processing system designs (cf. Table 4.1) reflecting different combinations of graph and delta formats. Specifically, we evaluate CSR+log, VoV+log, VoV+bitmap, and multi-version-array. They mirror design choices made in PowerGraph, Giraph [1], GraphPool [37], and LLAMA [45], and are abbreviated to *csr*, *log*, *bitmap*, and *m-array*, respectively.

Workloads. Table 4.1 summarizes the datasets [11,12] and algorithms. The Facebook [71] and GitHub graphs are collected as dynamically evolving graphs. The remaining five graphs are collected as static graphs [11, 12], for which deltas need to be created. Since deltas among consecutive versions are commonly within 1% of the graph size [53], we vary the delta size from 0.01% to 1%. We select $\delta = 0.1\%$ as a middle ground and show most of the evaluation results with this configuration. The total number of cached versions

Table 4.1: Graphs, algorithms, and reference designs

dataset	V (M)	E (M)	description
Amazon08	0.7	5.2	similarity among books
Dblp11	1.0	6.7	scientific collaboration
Wiki13	4.2	101.4	English Wikipedia
Livejournal	5.4	79.0	friendship in LiveJournal social network
Twitter	41.7	1468.4	Twitter follower graph
Facebook	0.1	1.6	friendship in regional Facebook network
GitHub	1.0	5.7	collaboration in software development

algorithm	description
nop	access neighbor and return
bipart	max matching in a bipartite graph
cc	identify connected components
PageRank	compute rank of each vertex
sssp	single-source shortest path
tc	triangle count

ref. design	description
csr	use CSR graph and log delta
log	use VoV graph and log delta
bitmap	maintain union of neighbors in all versions in VoV graph and use bitmap delta
m-array	use multi-version-array graph/delta

varies broadly from 1 to 100. Unless otherwise specified, we use uniform add-only deltas: each delta consists of edge additions uniformly distributed over a graph. Graphs evolve linearly: version i is created by iteratively applying $\delta_{j,j+1}, j = 0 \ldots i-1$ to the root version (i.e., version 0). Version switching is local, in that all versions are within the range of $n\delta$ from the root version. Version switching is arbitrary. That is, the next version j is selected independently to the current version i, may precede or succeed i, and do not need to be consecutive to i.

Since machines with large memory and many cores become popular and affordable [50, 62], VT's evaluation focuses on single-host setting. The elimination of inter-host communication cost in graph processing stage further highlights the effect of neighbor access efficiency. All measurements except those related to the Twitter graph [38] are performed on a host with 8 3GHz vCPUs and 60GB memory. Twitter-related workloads run on a host

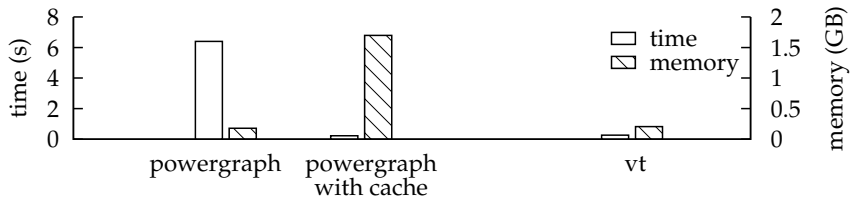

Figure 4.10: Comparison of VT and PowerGraph

with 32 2.5GHz vCPUs and 244GB memory.

Metrics. The requirements on extensibility, compactness, and access efficiency naturally lead to the use of time and memory consumption as two basic metrics. In addition, inspired by the resource-as-a-service model in the economics of cloud computing [4], we introduce a penalty function as a third metric: $p = (t_s + t_c)^{\alpha} \times m^{\beta}$. The penalty p is a function of the version switching time t_s, the computation time t_c, and memory consumption m. α and β are weights associated with time and memory resource. If the per-time-unit monetary cost is determined only by memory consumption, then assigning 1 to both parameters equates the penalty with the per-task monetary cost.[8] We use $\alpha = 1$ and $\beta = 1$ in our evaluation. When appropriate, we report penalty score p in the form of *utility improvement*: the improvement of VT over a reference system *ref* is calculated as $\frac{p_{ref} - p_{vt}}{p_{ref}}$.

Delta Preparation. For each system/workload setting, deltas corresponding to versions accessed in that workload are populated in memory, according to the delta design employed by that system, before the start of the workload. For VT, we employ a threshold-based policy (cf. Section 4.2.2.3), determining the delta format according to the number of neighbors in its source version and switching from Full to Sharing to Chaining as the number increases. We sample the threshold space for Full-Sharing and Sharing-Chaining transitions and report the lowest penalty score.

Comparison with PowerGraph. We evaluate the performance of PowerGraph by running

[8] Time and resource, nevertheless, do not always have the same weight. For instance, if graph processing resides along the critical path of a higher-level data analytics framework, then a user may assign a higher weight to α, willing to trade memory for sublinear speedup.

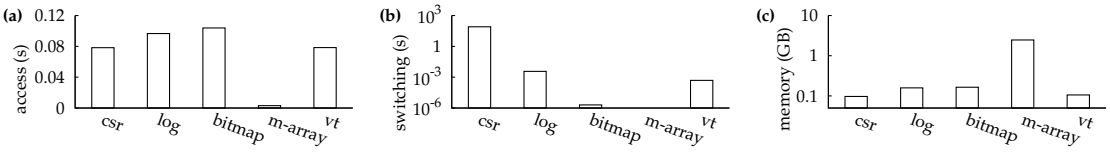

Figure 4.11: Comparison with existing graph/delta designs

SSSP on Amazon with ten 0.1% δs in two scenarios. First, we measure the performance of PowerGraph as-is, with a graph version loaded from persistent storage in its entirety at the beginning of each task. Second, we augment PowerGraph with full-version caching, storing each version in the working set as a full graph copy in memory.

Figure 4.10 shows that VT significantly outperforms PowerGraph in both scenarios. VT's processing speed is on a par with that of PowerGraph with full-graph caching and is 23x faster than that of PowerGraph without caching. This is due mainly to PowerGraph's substantial loading time when caching is disabled. VT's memory footprint is close to that of PowerGraph without caching (incurring a 15% overhead) and is only 12% of that of PowerGraph with full-graph caching—a 7.3x enhancement. Overall, VT improves utility by 86% and 95% over PowerGraph with and without caching, respectively.

Comparison with Multi-Version Designs. Figure 4.11 compares four multi-version designs with VT, executing nop on Amazon with ten 0.1% δs. *Csr* incurs prohibitive switching cost, due to CSR's low extensibility. It, nevertheless, yields the highest performance and has the smallest memory footprint. Both *log* and *bitmap* consume more memory than VT. *Bitmap* incurs a computation-stage penalty due to bitmap checking. *Log*'s switching cost is 7.4x that of VT.

Regarding *m-array*, its neighbor access time and version switching time are significantly shorter than the other designs. Its memory consumption, however, is much higher than the other. It is important to note that *m-array*'s superior performance is an outcome of efficient *implementation* of LLAMA, not a result of the multi-version-array design. This is because, after a version becomes ready for processing, all things being equal, *csr* should

yield the highest neighbor access performance for the nop workload. The difference between *m-array* and *csr* is then due to the framework-related overhead: *m-array* is measured with LLAMA and *csr*—as well as the other designs—is measured with PowerGraph-based implementation. Had we ported *m-array* to PowerGraph, its performance would be at best on a par with *csr*, and thus also close to VT.

M-array's high memory consumption is a result of the multi-version-array design. For Amazon with 0.1% δs, each version contains 5.2K new edges. Uniformly distributed, those edges affect 5.2K vertices' neighborhood. In LLAMA, with a 16-byte vertex record[9] and a 4KB page, the entire vertex record array for the root version spans 2.7K pages, which is also the expected number of pages affected when the 5.2K vertices with modified neighborhood are uniformly distributed. This yields a 100% memory overhead in terms of per-version vertex record array—because the entire 2.7K pages containing the root multi-version array need to be copied for each version—and a 21.5% overhead when the entire graph connectivity structure (with neighbor arrays) is considered. Such an overhead is prohibitively expensive for large graphs. In contrast, VT has a smaller footprint for the root version and, more importantly, incurs only a 0.6% per-version overhead for the graph connectivity structure in its Chaining mode.

Figure 4.11 confirms our expectation on the advantages and shortcomings of existing designs. Given *csr*'s low extensibility and *m-array*'s high memory consumption, we focus on comparing VT with *log* and *bitmap* for the rest of the evaluation.

Comparison with *log* and *bitmap*. Figure 4.12 summarizes the results comparing VT with *log* and *bitmap*, each with 10 δs of size 0.1%. VT consistently outperforms both systems in all but one case. Except the Twitter-SSSP workload, VT runs 2–17% faster in average per-version processing time and achieves 17–34% memory saving and 19–40% utility improvement.

Running SSSP over the Twitter graph, VT runs 88% faster than *log* but 19% slower

[9] A vertex record consists of version id, offset into the neighbor array, number of new edges for the current version, and an optional out-degree, each occupying 4 bytes.

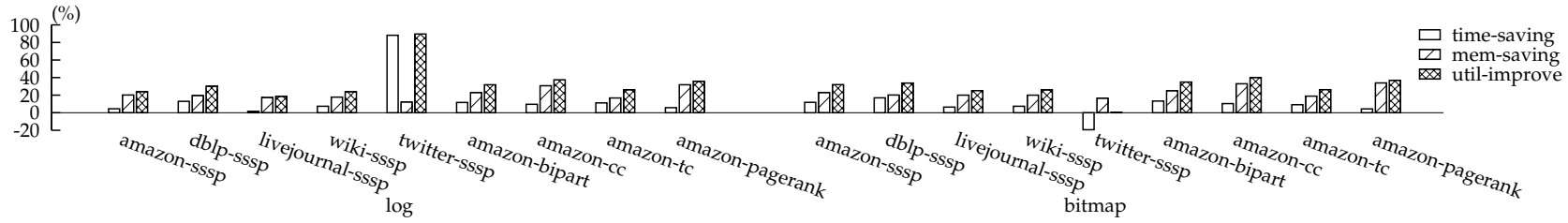

Figure 4.12: Comparison of VT with *log* and *bitmap* across all datasets and algorithms with 10 0.1% δs

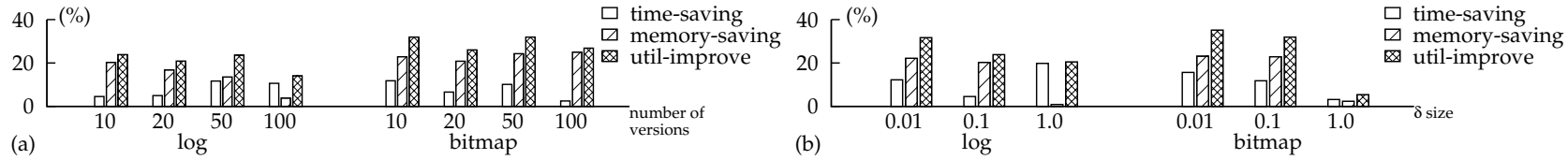

Figure 4.13: Varying number of δs and δ size. (a) Fixing $\delta = 0.1\%$ and varying number of versions between 10 and 100. (b) Fixing number of versions to 10 and varying δ between 0.01% and 1.0%.

than *bitmap*. This is because, given the size of the dataset, the configuration of the supporting hardware, and the characteristics of the algorithm, the difference in version switching dominates the overall processing efficiency. *Log* falls far behind VT, due to the former's need of log replaying during version switching. VT's delta application, although efficient and highly parallelized, is still a heavier-weight operation compared to *bitmap*. Combining time and memory consumption, the net effect is that VT outperforms *log* by 90% and is on a par with *bitmap* in utility improvement.

Varying Deltas. We compare VT with *log* and *bitmap* by executing SSSP on Amazon08, varying the size of delta from 0.01% to 1% and the number of deltas from 10 to 100. Fixing the delta size to 0.1% and varying the number of deltas from 10 to 100, we observe that VT's utility gain remains high with respect to *log* and *bitmap* (cf. Figure 4.13a). Compared to *log*, VT's memory saving reduces with the increasing number of deltas, because the memory consumption of the delta cache grows with the number of deltas, gradually neutralizing the benefit of the use of hybrid CSR. VT's gain due to the reduction of version switching time increases with the number of deltas, however. Overall, with these opposite trends, VT's utility gain remains high. Compared to *bitmap*, VT's memory saving remains high, because of *bitmap*'s need to maintain per-version bitmaps. VT's saving in processing time reduces, however, because the impact of *bitmap*'s saving in version switching time increases with the number of deltas, compensating for *bitmap*'s neighbor-access slowdown in the computation stage. The overall effect of these opposite trends is VT's constantly high utility gain with respect to *bitmap* across a wide range of versions.

Fixing the number of deltas to 10 and varying the size of delta from 0.01% to 1%, we observe that VT's utility gain gradually reduces (cf. Figure 4.13b). Compared to *log*, VT's gain peaks at $\delta = 0.01\%$, thanks to its efficient graph-delta representation. VT's gains for $\delta = 0.1\%$ and $\delta = 1\%$ are similar: larger deltas reduce VT's advantage in memory representation but amplify its reduction of version switching cost. Compared to *bitmap*, VT's utility gain remains high for $\delta = 0.01\%$ and $\delta = 0.1\%$, but drops significantly for

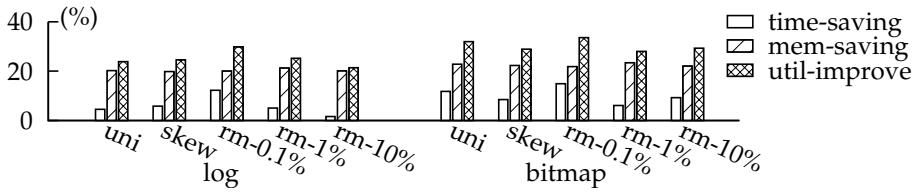

Figure 4.14: VT with uniform, skewed, and add/rm deltas

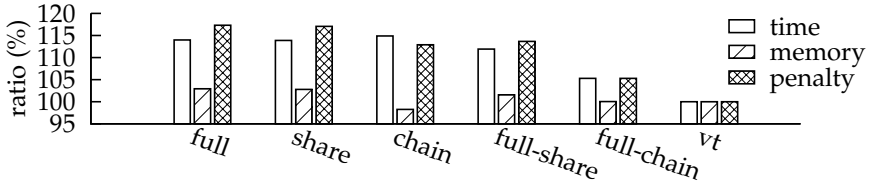

Figure 4.15: Effectiveness of optimization, with 10 0.1% δs (normalized to VT)

$\delta = 1\%$. Note that, the maximum distances among versions—in terms of dissimilarity—are the same for 100 0.1% δ (in Figure 4.13a) and 10 1% δ (in Figure 4.13b). Yet, VT's gain with respect to *bitmap* is much higher in the former case. This is because VT's memory saving is more significant when *bitmap* needs to maintain a larger number of per-version bitmaps in order to track the neighbor-version relation.

Skewed and Add/Remove Workloads. Figure 4.14 compares VT's performance across three types of workloads, all with ten 0.1% δs. The first is a uniformly distributed add-only delta type, same as those used throughout the evaluation. The second is a skewed add-only delta, in which the probability of adding a new edge to a vertex is proportional to the latter's degree in the root version. The third is a mixed add/remove delta type, with each delta maintaining a removals/total operations ratio varying from 0.1% to 10%. VT consistently outperforms *log* and *bitmap* in all the three workloads.

Effectiveness of Optimization. Figure 4.15 summarizes the effectiveness of Sharing and Chaining. Reusing the workload of SSSP-Amazon with ten 0.1% δs, we first enforce a fixed delta format, measuring the performance of Full, Sharing, and Chaining individually. We then combine Full and one of the two optimization approaches and report the minimum achievable penalty. All results are then normalized to those of VT. The effec-

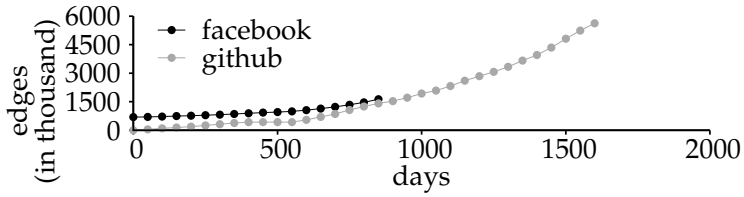

Figure 4.16: Evolution trends of a regional Facebook friendship graph and a GitHub collaboration graph

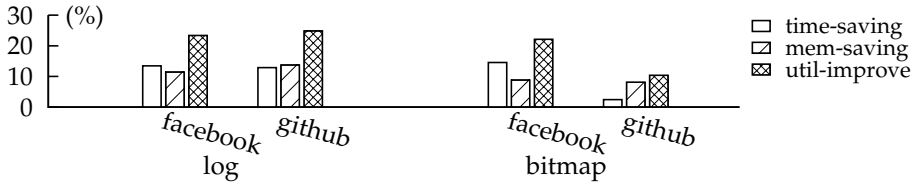

Figure 4.17: Performance of VT, *log*, and *bitmap* on Facebook and GitHub graphs

tiveness of Sharing and Chaining is demonstrated by the superiority (in terms of penalty) of a combined delta preparation strategy (e.g., Full-Sharing) to both approaches when applied individually (e.g., Full and Sharing). It is also demonstrated by VT's superiority—with all three delta formats combined—to the five alternatives.

Realistic Evolving Workloads. We compare VT with *log* and *bitmap*, using two 10-version graphs generated from the evolving Facebook friendship and GitHub collaboration graphs,[10] respectively. Figure 4.16 shows their evolution trends. Specifically, we choose 10 consecutive days towards the end of the collected periods for the two graphs[11] and combine newly established friendship/collaboration relations in each day into a delta. Friendship/collaboration relations existing before that 10-day period then form the root versions of the two graphs. Figure 4.17 shows that VT outperforms both *log* and *bitmap* when executing SSSP over these two graphs, improving utility by 10.38–24.83%.

[10]In the GitHub graph, the collaboration (i.e., edge) between two users (i.e., vertices) is established when they start to work on at least one shared repository. The initial state of the graph is set to empty. Its evolution spans between March 2011 and July 2015. We generate this graph via the use of GitHub API [2] and GitHub Archive [3].

[11]For the Facebook graph, daily delta size reduces drastically towards the end of the collected period, which might be caused by limitations of the collection method. We avoid those anomalies when creating the multi-version graph for evaluation.

4.4 Related Work

In this section, we discuss additional related work within the broad scope of multi-version graph processing but not dedicated to in-memory graph/delta design in the context of arbitrary local version switching.

Kineograph [21] and GraphChi [39] support streaming processing—a special case of multi-version graph processing where version switching is always forward and versions are only processed once. Such a solution is insufficient for the general multi-version scenario, where switching is arbitrary and a cached version is repeatedly accessed by multiple algorithms. Chronos [31] reorganizes the memory layout of multiple versions to improve cache utilization. FVF [53] proposes a specialized multi-version processing framework and demonstrates superior performance with several graph algorithms refactored for that framework. DataHub [8] provides a version control system supporting collaborative data analysis over multi-version datasets. Bhattacherjee et al. [9] investigates the tradeoff between storage space and recreation speed of a dataset version, focusing on organizing versions on disk instead of data structure redesign for in-memory caching.

None of the prior work systematically investigates the multi-version caching design space, which has significant impact on the overall performance and memory usage. VT takes a fundamental step in this direction.

4.5 Conclusions

In this chapter, we conducted a systematic investigation of the caching design space in multi-version graph processing scenarios, decomposing it into three dimensions: neighbor access efficiency, extensibility, and compactness. Our solution, Version Traveler, balances requirements from all three dimensions, achieving fast and memory-efficient version switching. It significantly outperforms PowerGraph and is superior to four multi-version reference designs.

CHAPTER V

Kairos: Efficient Parallel Multi-Version Graph Processing via State Consolidation

In this chapter, we first use a concrete example to demonstrate state sharing opportunities in parallel multi-version graph processing and then discuss the challenges related to in-memory graph state consolidation and the elimination of redundant state transition. We then present Kairos, a parallel multi-version processing system that addresses the above issues with dynamic state splitting. We also discuss the use of version collapsing to mitigate the version propagation effect—a side effect of dynamic state splitting.

5.1 Opportunities and Challenges

5.1.1 Opportunities

Figure 5.1 showcases the resource and computation sharing opportunities by applying the PageRank algorithm on the Amazon graph [11, 12]. We create a multi-version graph by using the original dataset as the base graph and randomly adding/removing a certain percentage of edges per version (ranging from 0.00001% to 10%). We then run PageRank on 10 versions of the graph in parallel for 20 iterations. We use the number of updates to vertices (i.e., the number of activation of vertex-centric computation) as the metric for the computation cost and the number of vertex versions (i.e., the sum of per-vertex versions

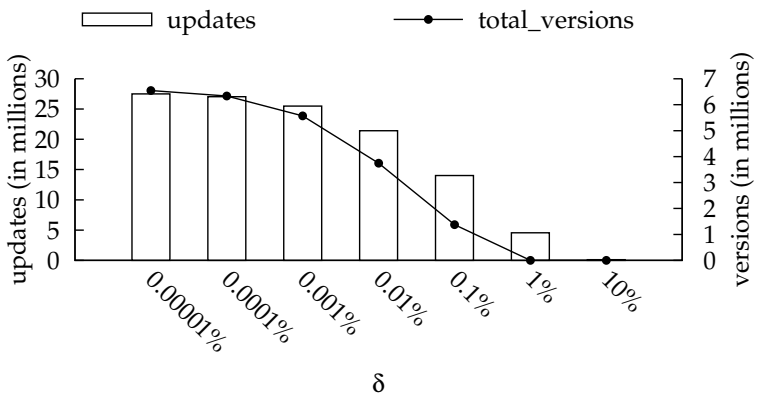

Figure 5.1: Sharing opportunities regarding resource and computation reuse.

over all vertices) as the metric for the memory usage. For example, the leftmost data points show that, when each non-default version differs from the default version by 0.00001% of the edges—only one edge in the case of Amazon, then processing the multi-version graph on a parallel multi-version processing system that can explore the resource and computation sharing opportunities would lead to the reduction of 27.5 million updates to vertex states and the elimination of 6.5 million copies of vertex states, compared with the same task processed in parallel on a single-version system. From Figure 5.1, we observe that (*i*) there exist substantial sharing opportunities when multiple similar graph versions are processed and (*ii*) the sharing opportunities in both computation and memory reuse diminish when graph versions become significantly different from each other.

5.1.2 Challenges

We discuss two major challenges in exploiting the sharing opportunities in multi-version graph processing: graph representation and version tracking.

5.1.2.1 Graph Representation Challenges

Not surprisingly, multi-version graph processing needs to properly represent graph versions on disk and in memory. On disk, multi-version graph processing requires the split of a graph version into a base graph and a delta file representing the difference between

the given version and the base graph. This is supported to some degree by existing graph processing systems (e.g., GraphLab only allows additions, but not deletions, of vertices and edges inside a graph delta file). As for the in-memory representation, multi-version graph processing requires a compact representation that ideally incurs only marginal overhead for each additional version atop the base graph. In addition, such a multi-version representation should minimize the memory access overhead with respect to that in a single-version graph. These two requirements lead to a common trade-off between performance and resource consumption. A compact representation, such as a tree-style structure, reduces memory consumption but increases the memory access cost. A structure that facilitates per-graph-version random access, such as a per-graph-version vector for vertices, maintains the memory access speed as in a single-version processing system, but incurs significant memory overhead.

5.1.2.2 Version Tracking

Version tracking refers to the bookkeeping of versioned states in a multi-version graph. Specifically, given a vertex and a version number, a multi-version processing system needs to be able to extract the vertex state related to that version, including its data fields and in/out edges.

Assume the use of a compact memory representation where vertex/edge states of non-default versions that are identical to those of the default version are consolidated. For any non-default version, version tracking involves determining whether or not each vertex shares the state of the corresponding vertex in the default version. This is straightforward (and static) if (*i*) graph algorithms are deterministic in how they modify vertex state, and (*ii*) for each vertex computation, the input state does not contain more versions than the center vertex. For most graph algorithms, however, there is no guarantee on the identity between the version information of the vertex state and that of its input state. The per-vertex version information thus gradually diverges from its on-disk counterpart during the

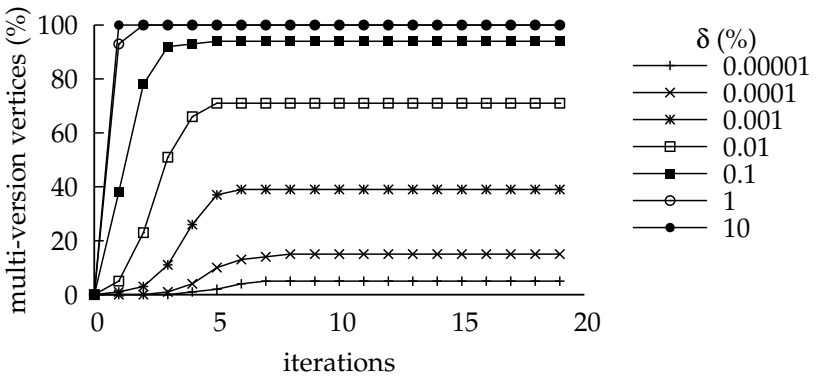

Figure 5.2: Version propagation effect. Apply the PageRank algorithm on the Amazon graph. Randomly modify 0.00001%-10% of edges per graph version (shown in the legend). Compute 10 versions in parallel. Plot the increase of the percentage of multi-version vertices (i.e., outcome of version propagation) along the execution.

course of per-vertex computation, leading to *version propagation*.

Figure 5.2 illustrates the effect of version propagation by applying the PageRank algorithm on Amazon. We run PageRank on 10 versions of the graph in parallel and observe the increase of multi-version vertices with the progress of the execution. From Figure 5.2, we observe that (*i*) vertices become multi-versioned along the execution of the algorithm, demonstrating the version propagation effect, and (*ii*) version propagation is positively correlated with the difference among graph versions.

5.2 Kairos

Kairos is a multi-version graph processing system that targets graph state consolidation in multi-version graph processing, as well as the elimination of identical state transitions.

5.2.1 Overview

Kairos extends the GAS model (cf. Algorithm 10) introduced in PowerGraph [26] in order to support multi-version graph processing. It also extends PowerGraph's graph representation and introduces version management to handle dynamic version tracking during

Algorithm 10 Per-Vertex Computation in Kairos

```
// gather-apply-scatter for center vertex V_u
// gather
for all i in versions do
  for all v in neighbors do
    g_i ← g_i + gather(V_{u_i}, E_{(u,v)_i}, V_{v_i})
// apply
for all i in versions do
  V_{u_i} ← apply(V_{u_i}, g_i)
// scatter
for all i in versions do
  for all v in neighbors do
    scatter(V_{u_i}, E_{(u,v)_i}, V_{v_i})
```

graph computation. Kairos reuses the vertex-cut technique for graph partitioning. The scheduling of vertex-centric computation is orthogonal to parallel processing of multiple graph versions. Kairos currently supports synchronous computation scheduling.

5.2.2 Programming Interface

Kairos maintains a PowerGraph-compatible programming interface and supports the gather-apply-scatter (GAS) model. Algorithms expressed in this model can be executed on Kairos with minor modifications over multi-version graphs.

We extend the original GAS abstraction with one additional interface that improves the efficiency of algorithms when processing multi-version graphs. Specifically, we define an "approximately equal" operator for the gather type, which is used for collapsing similar versions into the default version. The default behavior is to always return false, effectively disabling version collapsing. Users can enable version collapsing by overriding this function with an algorithm/gather-type-specific implementation. Kairos, in such a case, invokes the function at the end of the apply phase, folding non-default versions back to the default version when it is appropriate to reduce memory footprint and mitigate version propagation.

5.2.3 Graph Representation

We assume the existence of a common base graph from which all graph versions evolve. We have two design goals regarding Kairos's graph representation layer: (*i*) making the processing of single-version data fast—presumably the common case in large multi-version graphs—and (*ii*) compactly representing multi-version graphs. Take the representation of vertex data as an example (cf. Figure 5.3). Kairos uses a vector to store vertex data in the base version. This achieves the first goal: data access of the base version is as fast as single-version processing systems. When multiple versions exist for a given vertex, Kairos dynamically allocates a map and stores the mapping from version numbers to their corresponding vertex data. All pointers to these per-vertex maps are maintained in another vector addressed by vertex id. A `NULL` pointer in that vector indicates that the corresponding vertex remains single-versioned; that is, all versions share the same copy of vertex data.[1] The version-to-value mapping achieves the second goal: if a version shares the same state as the base version, then it is eliminated from the mapping.

We follow the same approach to extending other data structures, such as edges and messages. As for graph topology, we use the compressed sparse row (CSR) representation for vertices with single-version neighborhood and fall back to maps of versioned adjacency lists for multi-version vertices.

5.2.4 Version Splitting

Version management is the centerpiece of a multi-version graph processing system. The key idea is to (*i*) split versions, when necessary, to guarantee correctness of the computation and (*ii*) collapse versions, when appropriate, to mitigate the negative impact of version propagation on system performance. We focus on version splitting in this section.

Graph states evolve with the computation. So do the ways in which they are shared or unshared. Take vertex state as an example. As execution progresses, a single-version vertex

[1] Our current implementation also explicitly maintains the single/multi-version indicator in a bitset.

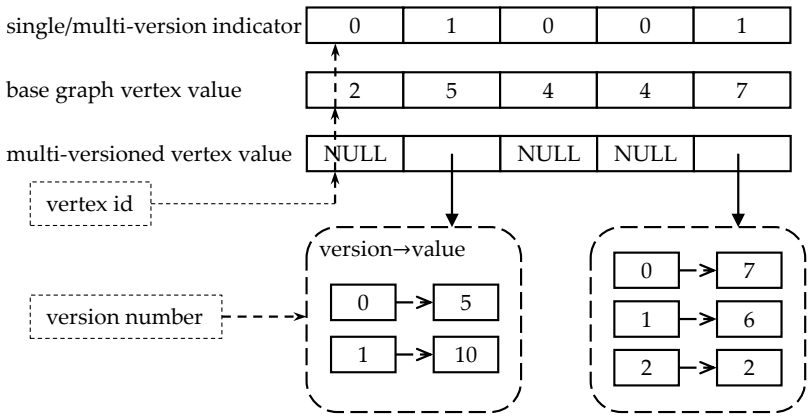

Figure 5.3: Illustration of vertex related data structure in Kairos. The single/multi-version indicator, the vertex value vector for the base graph, and the pointer vector for multi-version version to vertex data mapping are all addressed by vertex ids. The per-vertex version to data mapping uses version numbers as keys.

becomes multi-version when the vertex data related to at least one non-default version diverges from that in the default version. A multi-version vertex becomes single-version when all non-default versions converge to the state of the default version. In general, the version-to-value mapping may be inserted or removed from the multi-version vertex data map, depending on how many non-default versions can share the default value.

Version splitting requires copy-on-write (CoW) of vertex data. While kernel-mode page-level CoW can be efficiently achieved via standard MMU-based page protection, user-mode software systems, such as Kairos, need to rely on software approaches. Kairos achieves CoW of vertex data through inference. With the requirement that a vertex-centric graph algorithm be deterministic, the result of a vertex computation is determined only by the input. Thus, we can effectively infer the version of the output data based on that of the input data.

For example, suppose an algorithm specifies that, in each iteration, the center vertex updates its value on its own, as well as the values of its incoming neighbor vertices and incoming edges. For a given version i, the updated vertex data can be shared with the default version if the i-th version of all input states shares with their corresponding default

version. If such a condition does not hold and if the *i*-th version of the center vertex shares the default value before the update, then Kairos splits the *i*-th version of the center vertex from the default value, allocating its own memory area. The inference happens before the vertex computation so that, at the time of vertex update, the *i*-th version has already been allocated with its own copy of the vertex state.

Inference-based version splitting is applied to all data versioned by Kairos, including data and metadata (e.g., number of in/out edges) of vertices, edges, graph topology, as well as computation state, such as messages and gather values.

5.2.5 Version Probing and Caching

Given a non-default version, a common operation in per-vertex computation is to determine whether that version shares the state of the default version for a certain data type. This common operation is abstracted as an internal *version probing* interface for all data structures versioned by Kairos. Version probing can incur a non-trivial overhead to the per-vertex computation, in particular if the per-vertex computation itself is relatively lightweight. The overhead is exacerbated by the fact that a center vertex commonly needs to perform version probing for all its incoming/outgoing neighbor vertices and/or edges.

To speed up version probing, we implement a *version cache*, effectively collecting version information in a batch mode. For simplicity, in the rest of the section, we focus only on vertex-related version probing. The same mechanism is used for other data types, such as edges and gather values. Each vertex, upon entering one of the GAS phases, first populates a version cache indicating whether each non-default version of the vertex can share with the default version, given the version information of all its input states. Kairos then skips the computation phase all-together for those sharable versions (i.e., cache hits in the version cache), eliminating redundant version probing and per-vertex computation.

To understand the benefit of batched version probing, we compare the complexity of version cache population and version probing without caching. Suppose the state of the

center vertex is a function of its incoming neighbor vertices. Then, version cache population has $O(n)$ complexity, where n is the average number of per-vertex non-sharable non-default versions in the incoming neighbors. The complexity of version probing without caching is $O(m \log n)$, where m is the number of versions participating in the current computation phase and n the same as the cache population case. Version caching is found to be always beneficial except for small m values.

5.2.6 Version Collapsing

Kairos also supports version collapsing—the inverse function of version splitting—to boost system performance. In Kairos, due to the compact in-memory graph representation, both the computation cost and the memory footprint of a vertex computation are proportional to the number of non-default versions that do not share the state of the default version, instead of the total number of versions participating in the computation. Version collapsing effectively reduces the computation and memory overhead by folding identical or similar non-default versions into the default version.

Contrary to inference-based version splitting, version collapsing decisions are made by applying a user-defined "approximately-equal" comparator to determine the similarity of a non-default version and the default version.

5.2.7 Multi-Version Engine

Putting all together, we design Kairos's extended multi-version GAS engine.

Multi-Version Gather. The gather phase consists of two steps: (1) collecting information from the center vertex's neighbors via a user-defined gather function on each host and (2) merging them into one gather value via a user-defined `operator+=` for the gather type. Kairos extends `operator+=` from single-version state to the version-to-value mapping, merging two values if they have the same version number and inserting the version-value mapping existing in only one operand directly into the result. If the gather value of the i-th

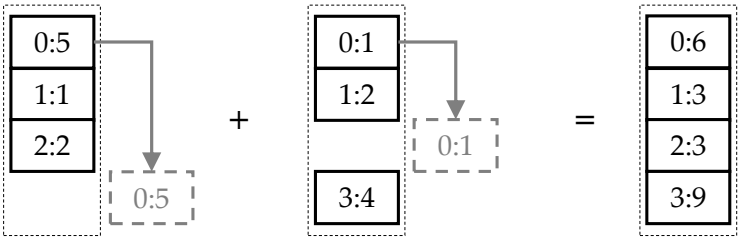

Figure 5.4: Illustration of multi-version gather. Each box with a dashed black border represents a multi-version gather value. Each box with a solid black border represents a version-value pair. Each gray box represents a version-value pair splitted during the gather operation. The generic operator += is extended from one gather value to a map of versioned gather values. Version split takes place, for example, in the left operand which does not have a dedicated copy for Version 3 before the gather operation.

version shares the default value before the gather in the left operand but the right operand has a dedicated copy of that version, then a version split takes place in the former before the addition (cf. Figure 5.4).

Multi-Version Apply. In the apply phase, for each center vertex, the multi-version gather value is folded into the multi-version vertex data. For each vertex, the user-defined apply function is called on all non-default versions, as indicated by its version cache. Inference-based version splitting for vertex data takes place when necessary (cf. Section 5.2.4). Versions that are inactive—that is, versions that participate in the current computation task but are deactivated for the current apply phase—incur additional complexity in the version splitting logic. For instance, suppose versions v_i, v_j, and v_k share the default version v_0 at the beginning of an apply phase. v_i and v_j are active and update the vertex data associated with the default version. v_k, in such a case, needs to be splitted from the default version in order to avoid false sharing and state corruption—a feature defined as *inactive version splitting* in Kairos. Kairos handles inactive version splitting after updating the active non-default versions and before updating the default version.

At the end of an apply phase, if the approximately-equal operator is defined for the vertex data type, then Kairos employs it for version collapsing.

Multi-Version Scatter. In the scatter phase, the user-defined scatter function is called for all non-sharable versions, as in the gather and apply phases. Multi-version messages are created when a vertex needs to signal its neighbors for their participation in the next superstep. Message versioning is necessary, since the per-vertex computation is activated at the granularity of versions (called *per-version computation activation* in Kairos). Message passing without versioning would force receiving vertices to activate all versions in the next superstep, potentially diverging the computation from algorithm specification and also incurring unnecessary overhead.

In order to support multi-version messaging, version unfolding takes place when at least one sharable version is active in the scatter phase. Specifically, thanks to the version cache, computation on all sharable versions is reduced to that on the default version. If the scatter function, when invoked with the default version, is about to send a message to its neighbors, then encoding only the default version in that message does not work. In effect, versions are locally maintained at each vertex. Consequently, the set of non-default versions sharable with the default version of one vertex may differ from that of its neighbors. Receiving a message containing the default version number, a vertex cannot correctly translate it to the set of non-default versions that are intended to be activated by the sender. It is the sender vertex's responsibility to explicitly unfold the version information locally and specifying the set of non-default versions that should be activated in a multi-version message.

5.3 Evaluation

In this section, we compare the performance of Kairos with PowerGraph. We first define evaluation metrics and then present the evaluation results based on micro-benchmarks with synthetic workloads as well as macro-benchmarks on realistic large graphs.

5.3.1 Metrics

We use three metrics to evaluate the performance of Kairos.

Execution Time. First, we calculate the speedup of Kairos over PowerGraph. Specifically, we define $speedup = \frac{t_{p_n}}{t_{m_n}}$, where t_{m_n} is the execution time of Kairos to run a multi-version computation task over n versions of a graph and t_{p_n} is that of PowerGraph. Assuming sequential processing of versions on PowerGraph, $t_{p_n} = t_{p_1} \times n$, where t_{p_1} is the average time for processing one version.[2]

Memory Consumption. We also compare memory usage of both approaches and report the memory overhead of Kairos with respect to PowerGraph. Memory overhead is defined as $overhead = \frac{m_{m_n}}{m_{p_1}}$, where m_{m_n} is the peak memory usage during the processing of a multi-version task with Kairos and m_{p_1} is that during the processing of each individual version on PowerGraph.

Cost. The elasticity and fine-grain billing [4] of cloud computing enables flexible trading and tuning between resource consumption and processing time. Users are charged according to the integral of resource consumption along the time domain. Assuming the existing of a cost per time-unit per resource-unit, time and resource become interchangeable regarding billing. Following this trend, we use a normalized performance gain (NPG) to quantify the performance gain of Kairos when jointly considering execution time and memory consumption.[3] The gain is calculated as the ratio of the cost of PowerGraph to that of Kairos, with the cost equating the product of execution time and memory consumption. That is, $NPG = \frac{speedup}{overhead} = \frac{nt_{p_1}m_{p_1}}{t_{m_n}m_{m_n}}$. An NPG larger than one indicates that Kairos outperforms PowerGraph.

[2] Given PowerGraph's version unawareness, its instance processes one version at a time. Kairos, in contrast, can process multiple versions in parallel.

[3] Note that, in defining NPG, we only consider the memory usage and ignore other resources, such as CPU and network I/O. This relative gain makes PowerGraph more favorable. For a fair comparison, we should also reduce the number of CPUs used by PowerGraph according to the measured memory overhead, i.e., $\frac{1}{overhead}$ of the CPUs used by Kairos. This, nevertheless, requires retrofitting CPU configuration after the measurement of memory overhead, the latter of which will then be affected by CPU resource re-allocation. We choose to use the memory overhead as the single factor for resource-usage comparison due to its dominant influence in in-memory processing of large graphs.

5.3.2 Microbenchmark

We implement a *no-op* microbenchmark algorithm to evaluate the overhead related to version management in Kairos. In each superstep, all vertices are activated, each gathering along its incoming neighbors and scattering to its outgoing neighbors. Gather/apply/scatter functions are all defined as no-op. The algorithm halts after 500 supersteps. For PowerGraph, the computation is performed on a single-version graph. For Kairos, its input is a non-default version identical to the default version (i.e., the same single-version graph with an arbitrary non-default version number to force Kairos to perform version inference). As a result, no version splitting or collapsing is triggered. The overhead of version management in this case stems from the need for Kairos to (*i*) determine, for each vertex, whether its vertex data, as well as other metadata such as in/out edges, are shared with the default version and (*ii*) fold/unfold the non-default version to/from the default version.

We also implement a *heavy-op* algorithm to isolate the effects of graph versioning on the performance of PowerGraph and Kairos. Each vertex contains an array of 10000 long integers. In the apply phase, we repeatedly compute the Fibonacci series and store them in the vertex for 10000 times. The algorithm halts after one superstep. The input graph is the same as that for the *no-op* algorithm so that version splitting or collapsing is not in use. The to-be-computed number of versions is set to 100. The choice of 100 versions, as with any other value, is intended to highlight the potential difference between PowerGraph and Kairos.

We run both algorithms on a synthetic graph with 3 thousand vertices and 1 million edges. Indegrees and outdegrees of all vertices follow uniform distribution. The measurement is performed on one blade of HP BladeSystem c7000, with 2 AMD Opteron 2214HE processors and 16 GB memory. The results, as shown in Table 5.1, clearly indicate the potential benefits and shortcomings of version management in graph processing.

For *no-op*, Kairos's graph computation time is significantly larger than that of PowerGraph. Such inefficiency mainly stems from two aspects. First, each multi-version data

Table 5.1: Performance comparison using the no-op and heavy-op algorithm. "perf" and "mem" refer to graph computation time (in seconds) and memory usage (in GB), respectively.

	no-op		heavy-op	
	perf	mem	perf	mem
Kairos	187.60	0.52	143.10	1.19
PowerGraph	22.75	0.48	14050	1.18
speedup/overhead	0.12	1.08	98.18	1.01
gain	0.11		97.21	

access in Kairos incurs two memory accesses, one to the single/multi-version indicator bitsets and the other to the data. In contrast, PowerGraph accesses the data directly. Second, in Kairos, each phase in the GAS model involves a version cache population step, preceding other user-defined vertex computation. When processing one single version, the version cache cannot speed up version probing. Populating the cache, however, is more costly than the no-op operation in each phase.

For *heavy-op*, Kairos significantly outperforms PowerGraph with marginal memory overhead. Since all versions are identical, Kairos only computes the default version and shares the result with all the other versions. PowerGraph, on the contrary, misses the reuse opportunity and needs to repeat the same computation for 100 times. This difference is linear to the number of versions. Moreover, the computation cost is dominated by generating the Fibonacci series. These two aspects explain the huge speedup of Kairos.

5.3.3 Macrobenchmark

We evaluate the performance of Kairos with the Amazon, Dblp, Wiki, and Livejournal datasets (cf. Table 4.1). We use three algorithms in our evaluation, PageRank, triangle count, and single-source shortest path. The same evaluation approach and metrics are employed in all experiments. In this section, we first present a thorough analysis on PageRank related results and then summarize results related to the other two algorithms.

We run PageRank on the four graphs while varying the difference between a non-

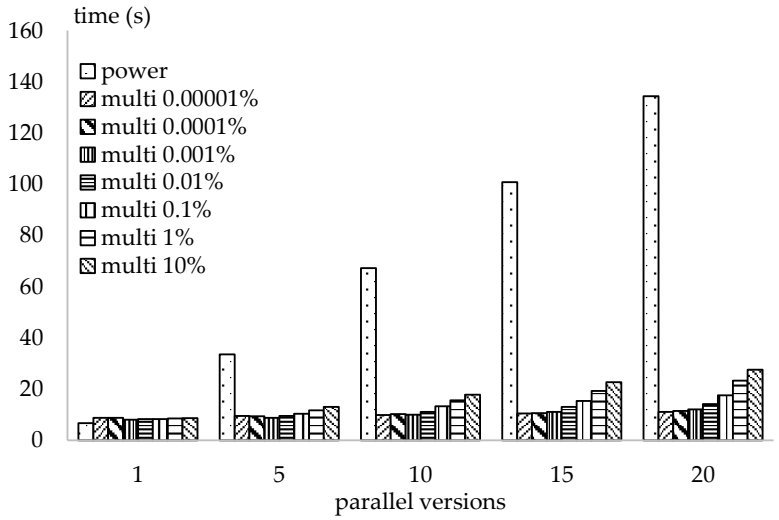

Figure 5.5: Performance of PowerGraph and Kairos when executing PageRank on Amazon. Each entry in the legend corresponds to a setting of the portion of the graph of a non-default version differing from the default version. For example, "multi 1%" indicates that each non-default version differs from the default by 1% of the size of the graph.

default version and the default version, as well as the number of versions processed in parallel. A non-default version differing from the default by $x\%$ is generated by randomly adding and removing $x\%$ of the total edges from the original graph. Vertices in all versions are identical. We define such a difference to be *per-version modifications*, which ranges from 0.00001% to 10% in our experiments. The accumulated difference of all non-default versions and the default version is thus the product of per-version modifications and the number of versions.[4]

All experiments with PageRank are executed on an 8-node Amazon EC2 cluster. Each node is a c3.8xlarge instance, with 32 vCPUs mapped to Intel Xeon E5-2680 v2 processors, 60 GB memory, and 10 Gbit Ethernet network. For PowerGraph, we run PageRank on the original graphs and report that as PowerGraph's average execution time for processing one version (t_{p_1}). We also measure the memory consumption of PowerGraph dur-

[4]Ideally, the accumulated difference is upper-bounded by the product of per-version modifications and the number of versions. For a given vertex, multiple versions may modify the base graph in the same way. The simplistic representation used in Kairos, however, makes it difficult to express the similarity among non-default versions. As a result, in our current implementation, the accumulated difference is equal to the product.

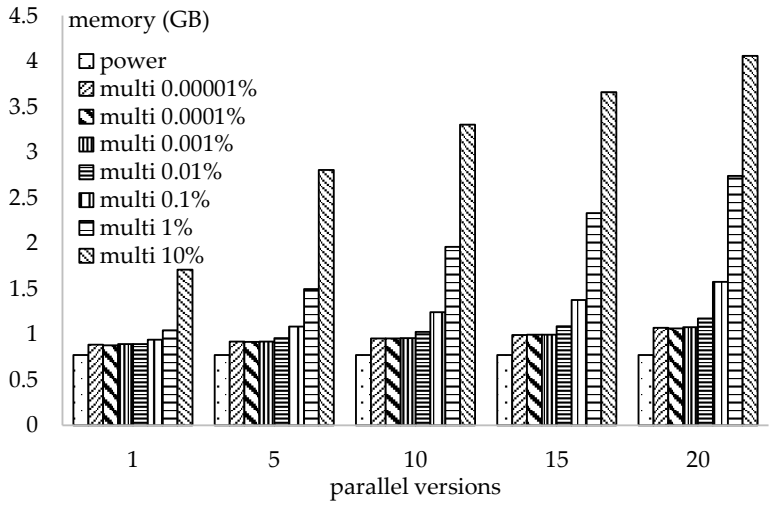

Figure 5.6: Memory consumption of PowerGraph and Kairos when executing PageRank on Amazon.

ing the processing of the default version and use that to represent the amount of memory PowerGraph consumes in processing any non-default version (m_{p_1}). For Kairos, we run PageRank in parallel on a multi-version graph and measure the execution time (t_{m_n}) and memory consumption (m_{m_n}). Results for each experiment setting are averaged over at least 3 measurements.[5]

Figure 5.5 shows the performance of PowerGraph and Kairos with the Amazon graph. Kairos consistently and significantly outperforms PowerGraph in terms of graph computation time for multi-version processing tasks by 2.57x to 12.06x. The speedup of Kairos increases with the number of versions processed in parallel. As expected, Kairos's processing time increases with the size of per-version modifications. Intuitively, the larger a non-default version differs from the default version, the less likely results of per-vertex computation can be shared between the two.

Figure 5.6 shows the memory consumption of PowerGraph and Kairos with Amazon. PowerGraph consumes a constant amount of memory, because it sequentially processes

[5]Note that, with our evaluation setting, there is no guarantee for either PowerGraph or Kairos to achieve optimal performance for any workload. In reality, for a parallel multi-version task, the placement and replication of input data, as well as instantiation of graph processing systems in a given computing environment, have significant impact on the overall performance.

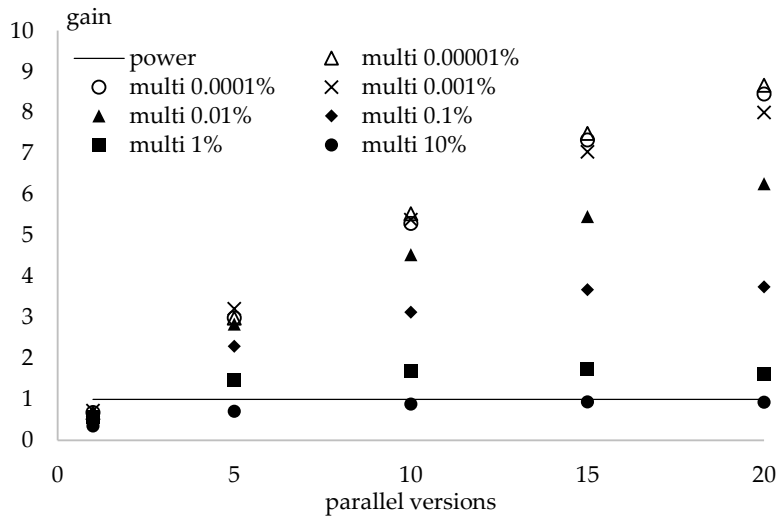

Figure 5.7: Normalized performance gain of Kairos . The baseline value is 1 (corresponding to PowerGraph). A value greater than 1 indicates the superiority of Kairos in multi-version graph processing.

each version. Kairos consumes more memory than PowerGraph, incurring an overhead between 1.19 and 5.27. The memory consumption increases with per-version modifications and parallel versions.

The normalized performance gain is shown in Figure 5.7. Clearly, when the graph is single-version, PowerGraph is superior to Kairos. With the increasing number of parallel versions, however, Kairos becomes superior, demonstrating its ability to efficiently process multiple versions in parallel, exploiting sharing opportunities on-the-fly. In fact, for all per-version modifications settings except 10%, Kairos demonstrates higher relative performance than PowerGraph. When each non-default version is 10% different from the default version, Kairos remains inferior to PowerGraph for all settings of parallel versions. This demonstrates the challenge imposed on Kairos by sizable inter-version differences.

The normalized performance gain varies significantly with workloads, as summarized in Table 5.2. For Amazon, Kairos remains superior until the per-version modifications amount to 10% of the original graph size. Results for Dblp show similar trends with those for Amazon 08. For Wiki and Livejournal, however, Kairos can only outperform PowerGraph when non-default versions are almost identical to the default version.

Table 5.2: Normalized performance gain of Kairos running Pagerank.

per-ver-mods %	parallel versions															
	Amazon				Dblp				Wiki				Livejournal			
	5	10	15	20	5	10	15	20	5	10	15	20	5	10	15	20
0.00001	2.98	5.53	7.49	8.67	2.58	3.47	4.23	4.60	0.95	1.05	1.01	0.87	1.13	1.33	1.33	1.16
0.0001	2.99	5.30	7.33	8.45	2.32	3.20	3.72	3.91	0.90	0.99	0.94	0.79	1.06	1.22	1.20	1.02
0.001	3.21	5.39	7.04	8.00	2.17	2.93	3.31	3.23	0.87	0.92	0.85	0.70	1.00	1.10	1.06	0.90
0.01	2.84	4.53	5.46	6.26	2.05	2.67	2.89	2.72	0.79	0.84	0.76	0.63	0.90	0.99	0.95	0.78
0.1	2.30	3.13	3.67	3.74	1.68	2.01	2.09	2.00	0.58	0.53	0.45	0.36	0.74	0.70	0.62	0.51
1	1.48	1.70	1.73	1.63	1.08	1.09	1.04	0.94	0.26	0.22	0.19	0.18	0.34	0.28	0.24	0.20
10	0.71	0.88	0.94	0.93	0.46	0.48	0.49	0.45	0.17	0.17	0.15	0.13	0.15	0.16	0.15	0.13

Table 5.3: Normalized performance gain of Kairos running Triangle Count.

per-ver-mods %	parallel versions															
	Amazon				Dblp				Wiki				Livejournal			
	5	10	15	20	5	10	15	20	5	10	15	20	5	10	15	20
0.00001	3.58	6.68	9.50	11.64	3.51	6.52	8.94	10.66	2.92	5.56	7.87	9.74	2.98	5.58	7.77	9.23
0.0001	3.52	6.67	9.43	11.50	3.51	6.57	8.79	10.48	2.98	5.50	7.84	9.66	3.06	5.67	7.67	9.27
0.001	3.45	6.43	9.19	11.07	3.52	6.41	8.77	10.44	2.99	5.53	7.62	9.18	3.01	5.42	6.97	7.41
0.01	3.27	5.88	8.24	9.37	3.28	5.29	6.38	6.89	2.12	3.70	4.29	4.29	2.58	3.89	4.42	4.55
0.1	2.91	4.73	5.60	6.19	2.83	4.63	5.50	5.89	0.99	1.24	1.27	1.24	1.41	2.07	2.25	2.31
1	1.68	2.26	2.56	2.81	1.35	1.59	1.63	1.61	0.33	0.27	0.25	0.22	0.46	0.43	0.38	0.34
10	0.50	0.50	0.47	0.42	0.38	0.36	0.33	0.29	0.12	0.09	0.07	0.06	0.13	0.10	0.08	0.07

Table 5.4: Normalized performance gain of Kairos running SSSP.

per-ver-mods %	parallel versions															
	Amazon				Dblp				Wiki				Livejournal			
	5	10	15	20	5	10	15	20	5	10	15	20	5	10	15	20
0.00001	3.95	7.67	11.17	13.91	3.75	7.12	10.26	13.18	1.60	2.56	3.33	3.71	1.72	3.15	4.19	4.85
0.0001	4.00	7.64	11.02	14.00	3.82	7.21	10.33	13.01	1.52	2.54	3.46	3.68	1.76	3.21	4.07	4.88
0.001	3.24	6.44	7.90	9.70	3.74	7.05	10.40	12.93	1.54	2.64	3.34	3.62	1.69	2.73	3.32	3.47
0.01	2.41	2.76	2.73	3.40	3.67	6.76	9.81	11.17	1.57	2.13	2.46	2.44	1.18	1.23	1.18	1.00
0.1	1.98	2.81	2.91	2.95	3.39	4.94	6.14	5.79	1.22	1.66	1.72	1.57	0.83	1.00	1.00	0.92
1	1.80	1.85	1.62	1.44	1.72	1.47	0.91	1.02	0.31	0.26	0.30	0.28	0.40	0.31	0.24	0.18
10	1.02	1.07	0.96	0.83	0.51	0.59	0.61	0.54	0.16	0.16	0.17	0.14	0.12	0.10	0.08	0.06

Table 5.5: Version collapsing related performance gain and accuracy loss. For each combination of per-version modifications and parallel versions, we present the relative gain of version collapsing with respect to the results without using version collapsing, as well as the loss of accuracy, in the format "relative gain (%) / accuracy loss (%)."

per-ver-mods %	parallel versions							
	Wiki				Livejournal			
	5	10	15	20	5	10	15	20
0.00001	30.01 / 0.41	42.63 / 0.43	59.38 / 0.48	72.07 / 0.48	76.44 / 0.00	133.11 / 0.00	173.57 / 0.00	210.10 / 0.00
0.0001	14.65 / 0.11	20.79 / 0.12	30.90 / 0.13	37.11 / 0.14	39.64 / 0.37	64.91 / 0.37	82.91 / 0.36	124.24 / 0.30
0.001	8.62 / 0.06	16.35 / 0.07	19.69 / 0.07	30.88 / 0.07	25.53 / 0.26	41.88 / 0.28	50.69 / 0.29	67.39 / 0.31
0.01	7.47 / 0.03	13.52 / 0.03	16.49 / 0.03	20.13 / 0.03	14.90 / 0.09	20.08 / 0.11	21.25 / 0.13	33.82 / 0.14
0.1	6.21 / 0.01	6.66 / 0.01	11.97 / 0.02	12.33 / 0.02	6.40 / 0.03	10.57 / 0.04	12.11 / 0.04	15.03 / 0.04

With Wiki and Livejournal, the gain decreases with the increase of per-version modifications—a trend that we have observed with the Amazon workload. The other trend—that is, the increase of gain along with the number of versions processed in parallel—no longer holds for Wiki or Livejournal. Both workloads, instead, show higher gain for versions equal to 10 and 15 than that for versions equal to 5 and 20. In practice, the number of versions that can be processed in parallel by Kairos cannot be infinite. A theoretical limit for the maximum number of parallel versions, or more generally, the range of parallel versions in which Kairos is advantageous, can be calculated based on the amortized computation overhead, memory access slowdown, and memory footprint increase related to a multi-versioned vertex with respect to its single-versioned counterpart, as well as the version propagation effect—a function of the user-defined algorithm and the graph workload. Table 5.2 indicates that the sweet spot of Kairos for applying PageRank on Wiki and Livejournal resides between 5 and 20 of parallel versions, while our current experimental setting cannot fathom the limit of Kairos for Amazon, at least for small per-version modifications. Results for Dblp show the shift of the sweet spot from above 20 versions for minor per-version modifications to between 5 and 20 versions for more significant modifications.

The performance of Kairos also depends on algorithms. Tables 5.3 and 5.4 show the relative gain of running triangle counting and SSSP on Kairos. Since the version propagation effect is constrained by algorithm design in these cases, the range of per-version modifications where Kairos is superior expands significantly for Wiki and Livejournal. In addition, the performance of Kairos keeps increasing with the number of parallel versions in most cases for all four workloads, indicating the dependency of the sweet spot in the dimension of parallel versions on algorithms.

5.3.4 Effectiveness of Version Collapsing

Table 5.5 shows the effectiveness of version collapsing in mitigating the version propagation effect during the execution of PageRank. For each combination of per-version

modifications and parallel versions, we present the relative gain of version collapsing with respect to the results without version collapsing, as well as the loss of accuracy. Specifically, the relative gain is calculated as $\frac{perf_{collapse} - perf_{original}}{perf_{original}}$, $perf_{collapse}$ denoting the NPG with version collapsing and $perf_{original}$ denoting the NPG without version collapsing. We use the L^1 norm to quantify the loss of accuracy. Specifically, $loss_{accuracy} = \frac{\sum_{i=1}^{\#vertices} |rank_{i_{original}} - rank_{i_{collapse}}|}{\sum_{i=1}^{\#vertices} rank_{i_{original}}}$, with $rank_{i_{original}}$ denoting the rank of vertex i without version collapsing and $rank_{i_{collapse}}$ denoting the rank with version collapsing. The version collapsing threshold is set to be 1/10 of the tolerance value in PageRank. Table 5.5 indicates that an algorithm can be designed to be multi-version-processing friendly via Kairos's version collapsing interface. One can conveniently trigger the version collapsing support of Kairos, explicitly trading a minor loss of accuracy, when appropriate, for a potentially substantial performance gain.

Figure 5.8 shows a case study on the effectiveness of version collapsing regarding memory footprint reduction, comparing the variation of the total number of versions when running PageRank on Amazon, with and without version collapsing. The total number of versions in a multi-versioned graph in the example is calculated as the sum of per-vertex non-default non-sharable versions plus the number of default versions (i.e., the number of vertices). Specifically, if a given vertex has two versions holding state different from the default version, then it contributes three versions to the total number of versions. From Figure 5.8, we observe that version collapsing is highly effective in this case, keeping the multi-versioned graph in a condensed representation.

5.4 Discussion

Relation to Incremental Processing. Incremental graph processing systems (e.g., Kineograph [21]) share a similar design principle to a multi-version graph processing system: improving efficiency by reusing the result of a preceding computation stage. It is the def-

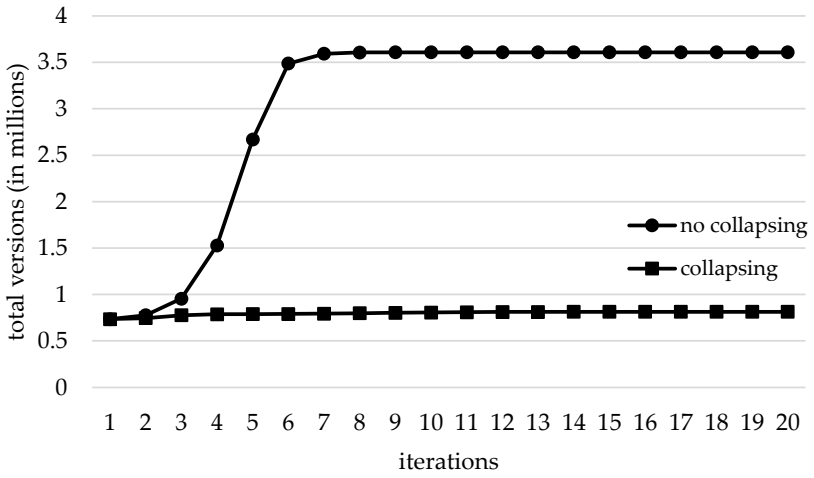

Figure 5.8: Case study on the effectiveness of version collapsing regarding memory footprint reduction (measured by the total number of versions).

inition of a computation stage that differentiates the two approaches (cf. Figure 5.9). In incremental processing, a computation stage is equivalent to a computation task and result reuse always takes place at the task boundary. Moreover, the computation stage at the graph level is aligned with that at the vertex level. A clear definition of the version-switch boundary in incremental processing simplifies system design. It requires, nevertheless, algorithms to be modified so as to reuse previous results at that boundary [22, 32]. Consequently, an incremental equivalent of an algorithm may not always be realizable. When they do, they may not produce results identical to their non-incremental counterparts.

In contrast, the computation stage in multi-version processing is only implicitly defined at the vertex level. When two versions of a vertex can no longer share a common state, the shared computation becomes the preceding computation stage and a version split with result reuse then takes place. Specifically, multi-version processing systems provide two key benefits over incremental processing systems: (*i*) the lifting of the requirement for incremental algorithms and (*ii*) implication, the guarantee on producing the same result as single-version processing systems. These benefits come at a cost: multi-version system can potentially have higher computation overhead and larger memory footprint (due to the difference between version switch and version split).

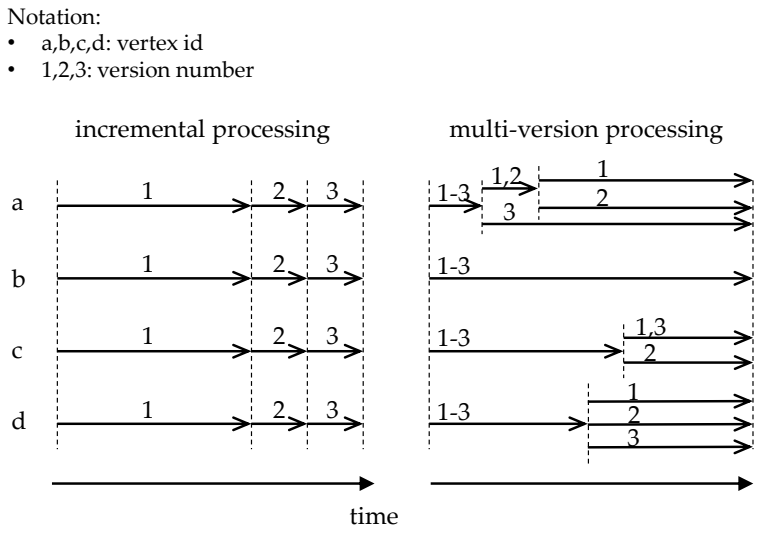

Figure 5.9: Workflow comparison between incremental and multi-version graph processing. Time progresses rightward. Vertex ids and version numbers are alphabetical and numerical, respectively. Vertical dashed lines indicate version switch/split boundaries.

Alternative Countermeasures to Version Propagation. To mitigate version propagation, we have experimented with version collapsing—an error-bound-based approach.

Another promising solution is to cluster versions according to a similarity metric and only share vertex/edge state among versions within a cluster. This approach is inspired by clustering in FVF [53] and for version indexing [37]. It is conceptually similar to the above error-bound approach: the latter is effectively a dynamic clustering approach that generates clusters at the level of vertices/edges along the course of computation.

A third tentative solution is to enable a multi-version processing system to switch from the state sharing mode back to the basic one-version-at-a-time approach when appropriate. Specifically, we could start the computation optimistically by launching all versions (or all versions within a cluster) in the state sharing mode. In the course of computation, if we determine that the severity of version propagation surpasses a threshold, then we pause the computation, checkpoint the current state for all versions, and then resume the computation one version after another. After the computation on a version is completed, we restore the system state using the checkpoint, and continue the processing of the next version.

A finer-grained variant of the above approach is to switch only a subset of versions off the state sharing mode and allow the rest to proceed in the state sharing mode. The selective switching may lead to better overall system performance but requires more sophisticated logic to determine the time for switching and the subset of versions to be switched off.

5.5 Related Work

Kineograph [21] targets incremental processing of graphs. Kairos differs from Kineograph in that the latter focuses on reusing the result from a previous computation and then continuing toward a different execution path. Only one execution is active at any given time. In contrast, Kairos processes multiple versions of a graph in parallel, progressing towards all paths as specified by a task. Conceptually, Kairos overlays multiple graphs processed by Kineograph and aligns them to the same starting point, forking new computation threads when necessary to guarantee correctness. GraphInc [16] achieves incremental graph processing without requiring the redesign of non-incremental algorithms—similar to a design goal in Kairos. GraphInc's use of memoization of all computation state during the execution of an algorithm, however, incurs significant cost. Chronos [31] is, to our knowledge, the first parallel multi-versioned graph processing system, capable of efficiently processing multiple temporal snapshots of one graph in parallel. By making one dedicated copy of vertex data for each snapshot, Chronos effectively avoids version management—one key design aspect in Kairos—and achieves high performance. Kairos, however, focuses on resource and computation sharing in parallel multi-versioned graph processing. The trade-off between more compact memory representation and the performance overhead due to version management, as well as the issue of version propagation, is thus well worth investigating. Kairos takes a first step in this direction.

5.6 Conclusions

In this chapter, we presented Kairos, a graph processing system optimized for parallel multi-versioned graph processing tasks. Kairos takes a first step in exploiting the sharing opportunities—in terms of both fine-grained per-vertex computation and in-memory graph representation—in cloud settings. We also identified version propagation as one key issue negatively affecting sharing opportunities across versions. We demonstrated the efficacy of Kairos in handling similar graph versions, its shortcomings in managing larger differences among graph versions, and the effectiveness of version collapsing as a countermeasure to the version propagation problem. The overall performance of Kairos demonstrates its potential as a promising solution for multi-versioned graph processing.

CHAPTER VI

Conclusions

In this thesis, we focus on the performance improvement of large-scale graph processing systems. We discuss three categories of deficiency in state-of-the-art graph processing systems and address each of them with a new systems design. We summarize the contibutions of this thesis and conclude with a discussion on future directions.

6.1 Contributions

We study the performance penalty of computation-communication coupling in the computation stage of single-version graph processing systems. We propose a new programming abstraction called Compute-Sync-Merge that supports locally-sufficient computation over vertex-cut, fully decoupling computation and communication. CSM enables the efficienct design of single- and multi-phase algorithms. Hieroglyph—our prototype implementation of a CSM-compliant system—outperforms state of the art by up to 53x.

Regarding sequential multi-version graph processing, we discuss the substantial overhead related to graph fetching, had each version of a multi-version graph been fetched individually, in its entirety. We address such deficiency with the integration of a graph caching layer into the multi-version graph processing workflow, enabling the construction of in-memory graph representation of a to-be-processed version based on graph contents present in the cache. Decomposing the caching design space into three dimensions, en-

compassing neighbor access effciency, extensibility, and compactness, we systematically investigate all of them. Version Traveler—our implementation of a graph processing system with a carefully design caching layer—significantly outperforms state of the art: running 23x faster with a 15% memory overhead compared to PowerGraph and achieving up to 90% improvement compared to several multi-version processing systems, when jointly considering processing time and resource consumption.

As for parallel multi-version graph processing, we study the presence of redundant graph state in memory and the computation cost related to redundant state transition. We design Kairos, a parallel multi-version graph processing system that consolidates redundant graph state and eliminates redundant state transition via dynamic state splitting. We also investigate countermeasures for mitigating the version propagation effect in dynamic version splitting.

6.2 Future Directions

This thesis focuses on optimizing the graph processing layer, which resides between supporting layers—such as systems software and hardware—and user-defined graph algorithms and applications. Higher performance can be achieved by carefully investigating the interactions of the graph processing layer with both the systems layer and the application layer.

At the systems level, efficient use of cache [31], RDMA [73], and GPU [47] may be critical for achieving high-performance graph processing. For example, enhancing cache utilization by prioritizing version-locality over graph-topology-related-locality can significantly speed up certain parallel multi-version processing tasks. Effective use of RDMA can improve the interleaving of computation and communication. GPU architectures can lead to significant gain in graph traversal with carefully devised parallelization strategies. Future graph processing systems need to explicitly take into consideration the features of

their supporting layers in order to further boost performance.[1]

At the application level, despite the wide adoption of vertex-centric programming paradigm, designing efficient vertex-centric graph algorithms remains difficult [58]. One promising approach to complement the efforts in algorithm design [35, 41] is to investigate the interaction between programming abstraction and algorithm design and, by co-designing the two, improve overall graph processing performance. Specifically, existing abstractions may need to be extended or restricted to better support certain categories of algorithms. Existing algorithms may need to be redesigned to fully unleash the power of state-of-the-art graph processing systems and carefully avoid their shortcomings.

Exciting domains are constantly emerging in real-time big data analytics, such as Internet of Things (IoT) and self-driving cars. Novel scenarios, such as the real-time analysis of massive data streamed from smartphones, smart watches, healthcare devices, home appliances, and cars, will impose new challenges to analytics frameworks. For example, how should massive traffic data be maintained to enable both offline analytics workloads and online decision making? How should fine-grained medical records generated by constant monitoring of numerous users be processed to help identify the propagation of a disease? Lessons learned from designing and improving graph processing systems, such as the ones presented in this thesis, can be applied to such new scenarios, resulting in innovative systems to efficiently address new challenges.

[1] This trend is observed in other domains of systems design, as well [10].

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Apache Giraph. `http://giraph.apache.org`. Retrieved in Apr. 2016.

[2] Github api. `https://developer.github.com/v3/`. Retrieved in Apr. 2016.

[3] Github archive. `https://www.githubarchive.org/`. Retrieved in Apr. 2016.

[4] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. The resource-as-a-service (raas) cloud. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Cloud Ccomputing*, HotCloud'12, Berkeley, CA, USA, 2012. USENIX Association.

[5] X. Amatriain and J. Basilico. Netflix recommendations: Beyond the 5 stars. `http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html`. Retrieved in Apr. 2015.

[6] Amazon. What is Amazon Kinesis? `http://docs.aws.amazon.com/kinesis/latest/dev/introduction.html`. Retrieved in Apr. 2015.

[7] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: Membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 44–54, New York, NY, USA, 2006. ACM.

[8] A. Bhardwaj, S. Bhattacherjee, A. Chavan, A. Deshpande, A. J. Elmore, S. Madden, and A. G. Parameswaran. DataHub: Collaborative Data Science & Dataset Version Management at Scale. In *7th Biennial Conference on Innovative Data Systems Research*, CIDR '15, 2015.

[9] S. Bhattacherjee, A. Chavan, S. Huang, A. Deshpande, and A. Parameswaran. Principles of dataset versioning: Exploring the recreation/storage tradeoff. *Proc. VLDB Endow.*, 8(12):1346–1357, Aug. 2015.

[10] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: It's time for a redesign. *Proc. VLDB Endow.*, 9(7):528–539, Mar. 2016.

[11] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web*. ACM Press, 2011.

[12] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[13] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, Apr. 1998.

[14] Y. Bu, V. Borkar, J. Jia, M. J. Carey, and T. Condie. Pregelix: Big(ger) graph analytics on a dataflow engine. *Proceedings of the VLDB Endowment*, 8(2):161–172, 2014.

[15] A. Buluç and J. R. Gilbert. The combinatorial blas: Design, implementation, and applications. *Int. J. High Perform. Comput. Appl.*, 25(4):496–509, Nov. 2011.

[16] Z. Cai, D. Logothetis, and G. Siganos. Facilitating real-time graph mining. In *Proceedings of the Fourth International Workshop on Cloud Data Management*, CloudDB '12, pages 1–8, New York, NY, USA, 2012. ACM.

[17] Q. Chen, S. Bai, Z. Li, Z. Gou, B. Suo, and W. Pan. Graphhp: A hybrid platform for iterative graph processing. 2014.

[18] R. Chen, X. Ding, P. Wang, H. Chen, B. Zang, and H. Guan. Computation and communication efficient graph processing with distributed immutable view. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 215–226, New York, NY, USA, 2014. ACM.

[19] R. Chen, J. Shi, Y. Chen, and H. Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 1:1–1:15, New York, NY, USA, 2015. ACM.

[20] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li. Improving large graph processing on partitioned graphs in the cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 3:1–3:13, New York, NY, USA, 2012. ACM.

[21] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 85–98, New York, NY, USA, 2012. ACM.

[22] S. Chien, C. Dwork, R. Kumar, and D. Sivakumar. Towards exploiting link evolution, 2001.

[23] A. Ching. Scaling apache giraph to a trillion edges. `https://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges/10151617006153920`. Retrieved in Apr. 2015.

[24] J. Cho, H. Garcia-Molina, T. Haveliwala, W. Lam, A. Paepcke, S. Raghavan, and G. Wesley. Stanford webbase components and applications. Technical Report 2004-34, Stanford InfoLab, 2004.

[25] DIMACS. 9th dimacs implementation challenge - shortest paths. `http://www.dis.uniroma1.it/challenge9/download.shtml`. Retrieved in Oct. 2015.

[26] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[27] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, OSDI '14, pages 599–613, Broomfield, CO, Oct. 2014. USENIX Association.

[28] Y. Guo, M. Biczak, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke. How well do graph-processing platforms perform? an empirical performance evaluation and analysis. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 395–404, Washington, DC, USA, 2014. IEEE Computer Society.

[29] M. Han and K. Daudjee. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proc. VLDB Endow.*, 8(9):950–961, May 2015.

[30] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin. An experimental comparison of pregel-like graph processing systems. *Proc. VLDB Endow.*, 7(12):1047–1058, Aug. 2014.

[31] 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. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '14, 2014.

[32] T. Hayashi, T. Akiba, and Y. Yoshida. Fully dynamic betweenness centrality maintenance on massive networks. *Proc. VLDB Endow.*, 9(2):48–59, Oct. 2015.

[33] X. Ju, D. Williams, H. Jamjoom, and K. G. Shin. Version traveler: Fast and memory-efficient version switching in graph processing systems. In *Proceedings of the 2016 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'16, Berkeley, CA, USA, 2016. USENIX Association.

[34] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09, pages 229–238, Washington, DC, USA, 2009. IEEE Computer Society.

[35] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo. K-core decomposition of large networks on a single pc. *Proc. VLDB Endow.*, 9(1):13–23, Sept. 2015.

[36] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 169–182, New York, NY, USA, 2013. ACM.

[37] U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 997–1008, Washington, DC, USA, 2013. IEEE Computer Society.

[38] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, New York, NY, USA, 2010. ACM.

[39] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association.

[40] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[41] Z. Li, Y. Fang, Q. Liu, J. Cheng, R. Cheng, and J. C. S. Lui. Walking in the cloud: Parallel simrank at scale. *Proc. VLDB Endow.*, 9(1):24–35, Sept. 2015.

[42] Y. Low. *GraphLab: A Distributed Abstraction for Large Scale Machine Learning*. PhD thesis, Carnegie Mellon University, 2013.

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

[44] Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-scale distributed graph computing systems: An experimental evaluation. *Proceedings of the VLDB Endowment*, 8(3), 2014.

[45] P. Macko, V. Marathe, D. Margo, and M. Seltzer. LLAMA: Efficient graph analytics using large multiversioned arrays. In *Proceedings of the 2015 IEEE International Conference on Data Engineering (ICDE 2015)*, ICDE '15, Washington, DC, USA, April 2015. IEEE Computer Society.

[46] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[47] D. Merrill, M. Garland, and A. Grimshaw. Scalable gpu graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, New York, NY, USA, 2012. ACM.

[48] A. Mislove, H. S. Koppula, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Growth of the flickr social network. In *Proceedings of the First Workshop on Online Social Networks*, WOSN '08, pages 25–30, New York, NY, USA, 2008. ACM.

[49] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. ACM.

[50] Y. Perez, R. Sosič, A. Banerjee, R. Puttagunta, M. Raison, P. Shah, and J. Leskovec. Ringo: Interactive graph analytics on big-memory machines. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1105–1110, New York, NY, USA, 2015. ACM.

[51] M. Pundir, L. M. Leslie, I. Gupta, and R. H. Campbell. Zorro: Zero-cost reactive failure recovery in distributed graph processing. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 195–208, New York, NY, USA, 2015. ACM.

[52] A. Quamar, A. Deshpande, and J. Lin. Nscale: neighborhood-centric large-scale graph analytics in the cloud. *The VLDB Journal*, pages 1–26, 2015.

[53] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng. On querying historical evolving graph sequences. *Proceedings of the VLDB Endowment*, 4(11):726–737, 2011.

[54] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 410–424, New York, NY, USA, 2015. ACM.

[55] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 472–488, New York, NY, USA, 2013. ACM.

[56] S. Salihoglu, J. Shin, V. Khanna, B. Q. Truong, and J. Widom. Graft: A debugging tool for apache giraph. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1403–1408, New York, NY, USA, 2015. ACM.

[57] S. Salihoglu and J. Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pages 22:1–22:12, New York, NY, USA, 2013. ACM.

[58] S. Salihoglu and J. Widom. Optimizing graph algorithms on pregel-like systems. *Proc. VLDB Endow.*, 7(7):577–588, Mar. 2014.

[59] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 979–990, New York, NY, USA, 2014. ACM.

[60] B. Shao, H. Wang, and Y. Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, New York, New York, USA.

[61] Y. Shen, G. Chen, H. Jagadish, W. Lu, B. C. Ooi, and B. M. Tudor. Fast failure recovery in distributed graph processing systems. *Proceedings of the VLDB Endowment*, 8(4), 2014.

[62] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, New York, NY, USA, 2013. ACM.

[63] Y. Simmhan, N. Choudhury, C. Wickramaarachchi, A. Kumbhare, M. Frincu, C. Raghavendra, and V. Prasanna. Distributed programming over time-series graphs. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 809–818, May 2015.

[64] Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, and V. Prasanna. Goffish: A sub-graph centric framework for large-scale graph analytics. In F. Silva, I. Dutra, and V. Santos Costa, editors, *Euro-Par 2014 Parallel Processing*, volume 8632 of *Lecture Notes in Computer Science*, pages 451–462. Springer International Publishing, 2014.

[65] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga. Arabesque: A system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 425–440, New York, NY, USA, 2015. ACM.

[66] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From "think like a vertex" to "think like a graph". *Proceedings of the VLDB Endowment*, 7(3), 2013.

[67] K. U. and A. Deshpande. Storing and analyzing historical graph data at scale. *ArXiv e-prints*, Sept. 2015.

[68] S. Vadapalli, S. R. Valluri, and K. Karlapalem. A simple yet effective data clustering algorithm. In *Proceedings of the Sixth International Conference on Data Mining*, ICDM '06, pages 1108–1112, Washington, DC, USA, 2006. IEEE Computer Society.

[69] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.

[70] R. K. Vinayak, S. Oymak, and B. Hassibi. Graph clustering with missing data: Convex algorithms and analysis. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2996–3004. Curran Associates, Inc., 2014.

[71] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi. On the evolution of user interaction in facebook. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Social Networks (WOSN'09)*, August 2009.

[72] G. Wang, W. Xie, A. J. Demers, and J. Gehrke. Asynchronous large-scale graph processing made easy. In *6th Biennial Conference on Innovative Data Systems Research*, CIDR '13, 2013.

[73] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou. Gram: Scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 408–421, New York, NY, USA, 2015. ACM.

[74] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen. Sync or async: Time to fuse for distributed graph-parallel computation. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 194–204, New York, NY, USA, 2015. ACM.

[75] W. Xie, G. Wang, D. Bindel, A. Demers, and J. Gehrke. Fast iterative graph computation with block updates. *Proc. VLDB Endow.*, 6(14):2014–2025, Sept. 2013.

[76] Y. Xu, V. Olman, and D. Xu. Clustering gene expression data using a graph-theoretic approach: an application of minimum spanning trees. *Bioinformatics*, 18(4):536–545, 2002.

[77] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proc. VLDB Endow.*, 7(14):1981–1992, Oct. 2014.

[78] D. Yan, J. Cheng, Y. Lu, and W. Ng. Effective techniques for message reduction and load balancing in distributed graph computation. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, New York, NY, USA, 2015. ACM.

[79] D. Yan, J. Cheng, M. T. Özsu, F. Yang, Y. Lu, J. C. S. Lui, Q. Zhang, and W. Ng. A general-purpose query-centric framework for querying big graphs. *Proc. VLDB Endow.*, 9(7):564–575, Mar. 2016.

[80] Z. Yu, H.-S. Wong, and H. Wang. Graph-based consensus clustering for class discovery from gene expression data. *Bioinformatics*, 23(21):2888–2896, 2007.

[81] P. Yuan, W. Zhang, C. Xie, H. Jin, L. Liu, and K. Lee. Fast iterative graph computation: A path centric approach. In *Proceedings of the International Conference*

for High Performance Computing, Networking, Storage and Analysis, SC '14, pages 401–412, Piscataway, NJ, USA, 2014. IEEE Press.

[82] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 45–58, Berkeley, CA, USA, 2015. USENIX Association.

[83] X. Zhou, P. Chang, and G. Chen. An efficient graph processing system. In L. Chen, Y. Jia, T. Sellis, and G. Liu, editors, *Web Technologies and Applications*, volume 8709 of *Lecture Notes in Computer Science*, pages 401–412. Springer International Publishing, 2014.

[84] B. Zong, X. Xiao, Z. Li, Z. Wu, Z. Qian, X. Yan, A. K. Singh, and G. Jiang. Behavior query discovery in system-generated temporal graphs. *Proc. VLDB Endow.*, 9(4):240–251, Dec. 2015.