UNIVERSITY OF CALIFORNIA
RIVERSIDE

Exploiting Asynchrony for Performance and Fault Tolerance in Distributed Graph Processing

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Keval Dinesh Vora

June 2017

Dissertation Committee:

Dr. Rajiv Gupta, Chairperson
Dr. Laxmi Bhuyan
Dr. Nael Abu-Ghazaleh
Dr. Zizhong Chen

The Dissertation of Keval Dinesh Vora is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

This dissertation would not have been possible without the support of many people across various walks of my life.

I would like to thank my Ph.D. advisor, Prof. Rajiv Gupta, for giving me an opportunity to work under him. I express my deepest gratitude to him for his constant support and enthusiasm. This dissertation is a culmination of his careful mentoring and guidance throughout my Ph.D.

I would also like to thank Prof. Guoqing (Harry) Xu for letting me work under him at UC Irivne. His mentoring helped me develop a different viewpoint on various things, many of which became a natural part of this dissertation.

I was fortunate to work under Dr. Chen Tian at Huawei US R&D Center. His unencumbered enthusiasm always motivates me. I would also like to acknowledge Prof. Iulian Neamtiu and Prof. Xiangyu Zhang for collaborations across various works.

I would like to thank my dissertation committee members for their valuable feedback and support: Prof. Laxmi Bhuyan, Prof. Nael Abu-Ghazaleh and Prof. Zizhong Chen.

I remain ever grateful to Prof. Rajiv Gandhi who was the key instigator in fueling my desire to pursue graduate studies. I am also grateful to all my teachers that I have had throughout my life.

I would like to acknowledge the support of National Science Foundation via grants CCF-1524852 and CCF-1318103 to UC Riverside.

I would like to express my gratitude to all my colleagues at UCR and UCI: Abbas, Aftab, Amlan, Arash, Bo, Bruce, Bryan, Changhui, Chao, Devashree, Farzad, Gurneet, Hongbo, Kai, Khanh, Lu, Mehmet, Panruo, Sai, Sankha, Vineet, Yan, Zachary and Zhiqiang.

Finally, I would like to thank my family and friends for their endless love, support and blessings throughout my life.

ABSTRACT OF THE DISSERTATION

Exploiting Asynchrony for Performance and Fault Tolerance
in Distributed Graph Processing

by

Keval Dinesh Vora

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2017
Dr. Rajiv Gupta, Chairperson

While various iterative graph algorithms can be expressed via asynchronous parallelism, lack of its proper understanding limits the performance benefits that can be achieved via informed relaxations. In this thesis, we capture the algorithmic intricacies and execution semantics that enable us to improve asynchronous processing and allow us to reason about semantics of asynchronous execution while leveraging its benefits. To this end, we specify the asynchronous processing model in a distributed setting by identifying key properties of read-write dependences and ordering of reads that expose the set of legal executions of an asynchronous program. And then, we develop techniques to exploit the availability of multiple legal executions by choosing faster executions that reduce communication and computation while processing static and dynamic graphs.

For static graphs, we first develop a relaxed consistency protocol to allow the use of stale values during processing in order to eliminate long latency communication operations by up to 58%, hence accelerating the overall processing by a factor of 2. Then, to efficiently handle machine failures, we present a light-weight confined recovery strategy that

quickly constructs an alternate execution state that may be different from any previously encountered program state, but is nevertheless a legal state that guarantees correct asynchronous semantics upon resumption of execution. Our confined recovery strategy enables the processing to finish 1.5-3.2× faster compared to the traditional recovery mechanism when failures impact 1-6 machines of a 16 machine cluster.

We further design techniques based on computation reordering and incremental computation to amortize the computation and communication costs incurred in processing evolving graphs, hence accelerating their processing by up to 10×. Finally, to process streaming graphs, we develop a dynamic dependence based incremental processing technique that identifies the minimal set of computations required to calculate the change in results that reflects the mutation in graph structure. We show that this technique not only produces correct results, but also improves processing by 8.5-23.7×.

Finally, we demonstrate the efficacy of asynchrony beyond distributed setting by leveraging it to design dynamic partitions that eliminate wasteful disk I/O involved in out-of-core graph processing by 25-76%.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recent advancements in data analytics model data as graphs to capture domain-specific data relationships, giving rise to an important class of graph processing problems. While various parallel graph algorithms have been developed to perform useful analyses, the large sizes of real-world graphs necessitate the use of clusters that provide the compute and memory capacity needed to efficiently process the large graphs.

Iterative graph algorithms compute values for vertices in the graph by starting with an initial set of values and iteratively computing new values until they stabilize to a solution (i.e., until convergence). While graph algorithms can be expressed in different ways, the *vertex-centric model* enables users to express computations for a single vertex, allowing them to *think like a vertex* by only looking at the neighborhood of the given vertex [86]. The simplicity of this model has led to its wide acceptance by various graph processing frameworks [84, 44, 73] which require users to only provide vertex functions, while the framework takes care about the rest of the processing details like, parallelization,

communication, synchronization, etc. Furthermore, this model has formed a basis for other models to easily express computations, like the popular *edge-centric model* where more parallelism can be directly exposed to the runtime system [103, 102].

Iterative processing in graph algorithms can be performed using synchronous and asynchronous processing models as described next. The synchronous or the *bulk synchronous parallel (BSP)* [126] model processes the graph in steps such that each step makes a single pass over all the vertices to compute new values based on the values of neighboring vertices from previous step. This model can be viewed as having disjoint computation and communication phases such that vertex values computed in a given computation phase are visible to the neighboring vertices via the subsequent communication phase. The *asynchronous model* [8, 20], on the other hand, allows vertex values to be independently computed based on available values from neighboring vertices. Hence, asynchronous model eliminates the need of synchronized steps which allows vertex computations to proceed without frequently coordinating with other computations.

Even though it is well known that asynchronous processing is faster than synchronous processing [84], the BSP model is a preferred choice mainly because of two reasons; first, it is difficult to develop and maintain an asynchronous processing framework; and second, it is easier to analyze the convergence and correctness of synchronous algorithms compared to their asynchronous counterparts. While availability of various asynchronous graph processing frameworks alleviates the effects due to the first reason, limited analysis of asynchronous execution has left asynchronous processing fairly unexplored, limiting its benefits in various contexts of graph processing.

Hence, this dissertation explores asynchronous processing in detail to understand its algorithmic intricacies and execution semantics, enabling us to improve asynchronous processing and make it easier to reason about asynchronous execution semantics while leveraging its benefits. We first specify the asynchronous processing model in a distributed setting by identifying key properties of read-write dependences and order of reads to expose the set of legal executions for asynchronous programs. And then, we develop key techniques to exploit the availability of multiple legal executions by choosing faster executions to accelerate the overall processing via reduction in both, communication and computation while processing static and dynamic graphs. Finally, we also demonstrate how the asynchrony can be exploited to improve out-of-core graph processing by eliminating wasteful disk IO.

1.1 Dissertation Overview

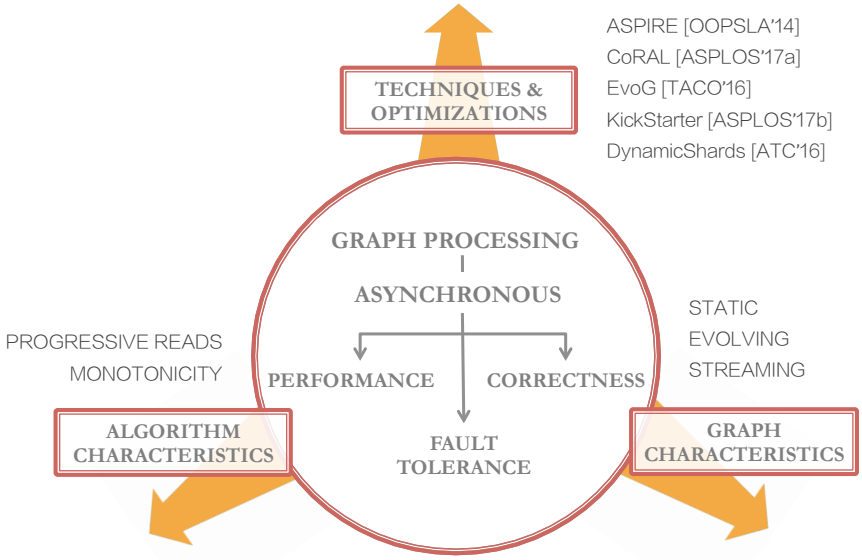


Figure 1.1: Dissertation Overview

In this dissertation, we exploit the inherent asynchrony present in various graph algorithms to further improve asynchronous graph processing. Figure 1.1 shows the overview of this dissertation. We develop novel processing techniques to improve the performance and fault tolerance of distributed asynchronous graph processing by carefully characterizing important algorithmic properties like *progressive reads semantics* and *monotonicity*. In doing so, we first address the challenges in processing static graphs, i.e., whose structure does not change over time, and then, tackle the issues in processing dynamic graphs.

1.1.1 Processing Static Graphs

Static graphs model relationships that do not change over time. Hence, the structure of the graph, i.e., the edges and vertices in the graph remain same throughout the execution of the graph algorithm. We first design a relaxed consistency model to tolerate long latency communication operations while processing static graphs and then develop a confined recovery strategy to quickly recover from machine failures during processing.

Relaxed Consistency Model

Various iterative graph algorithms (e.g., graph mining, graph analytics, PDE solvers) can be expressed via asynchronous parallelism by relaxing certain read-after-write data dependences. This allows the threads to perform computations using stale (i.e., not the most recent) values of data objects.

In a distributed environment, the graph is partitioned across a set of machines and each machine operates on the subgraph residing on that machine. Hence, the access latency experienced by threads in the system varies based upon the machine at which the

object is physically stored. Therefore the key to obtaining high-performance on a cluster is successfully tolerating the high inter-machine communication latency. On our *Tardis* cluster which has 16 compute nodes, the latency of accessing an object increases by 2.3× if instead of being found in the local software cache it has to be brought from a remote machine. We observe that to tolerate the long inter-machine communication latency, the asynchronous nature of algorithms can be exploited as follows. When a thread's computation requires an object whose stale value is present in the local cache, instead of waiting for its most recent copy to be fetched from a remote machine, the stale value can be used. By tracking the *degree of staleness* of cached objects in terms of the number of object updates that have since been performed, we can limit the extent to which use of stale values is permitted. However, fruitfully exploiting this idea is challenging due to the following:

- *Allowing high degree of staleness* can lead to excessive use of stale values which in turn can slow down the algorithm's convergence, i.e. significantly more iterations may be needed to converge in comparison to the synchronous version of the algorithm. This can wipe out the benefits of exploiting asynchrony.

- *Allowing low degree of staleness* limits the extent to which long latency fetch operations can be tolerated as the situation in which the cached value is too stale to be used becomes more frequent causing delays due to required inter-machine fetches.

We address the above challenge by designing a relaxed memory consistency model and cache consistency protocol that simultaneously *maximize the avoidance of long latency communication operations* and *minimize the adverse impact of stale values on convergence.* This is achieved by the following two ideas:

- First, we use a consistency model that supports bounded staleness and we set the permissible bound for using stale values to a high threshold. By doing so, we *increase the likelihood of avoiding long latency fetch operations* because if a stale value is present in the cache it is highly likely to be used.

- Second, we design a cache consistency protocol that incorporates a policy that *refreshes* stale values in the cache such that when a stale value is used to avoid a long latency operation, the likelihood of the value being minimally stale is increased. Thus, this refresh policy *avoids slowing down the convergence* of the algorithm due to use of excessively stale values.

Hence, significant performance improvements are obtained by effectively tolerating inter-machine fetch latency. Based upon such a relaxed memory consistency model, we present a vertex centric approach for programming asynchronous algorithms in a distributed environment. We demonstrate that for a range of asynchronous graph algorithms and PDE solvers, on an average, our approach outperforms algorithms based upon: prior relaxed memory models that allow stale values by at least 2.27× and Bulk Synchronous Parallel (BSP) model by 4.2×.

Confined Recovery

Fault tolerance in distributed graph processing systems is provided by periodically snapshotting the vertex/edge values of the data-graph during processing, and restarting the execution from the previously saved snapshot during recovery [86, 135]. The cost of fault tolerance includes: overhead of periodic checkpoints that capture *globally consistent snap-*

shots [19] of a graph computation; and *repeating computation* whose results are discarded due to the roll back during the recovery process. For synchronous graph processing systems, solutions that lower these overheads have been proposed. Pregel [86] performs *confined recovery* that begins with the most recently checkpointed graph state of the failed machine and, by replaying the saved inputs used by boundary vertices, recovers the lost graph state. Zorro [96] avoids checkpointing and directly recovers the lost graph state at the cost of sacrificing accuracy of computed solutions.

However, development of efficient fault tolerance techniques for asynchronous graph processing lags behind. To perform recovery, asynchronous processing systems roll back the states of *all the machines* to the last available snapshot and resume the computation from that point. This is because of inherent non-determinism in asynchronous processing which discards the possibility of reconstructing lost execution state using the saved inputs. Hence, fault tolerance in asynchronous graph processing systems has the following two drawbacks:

- *Redundant computation*: Since recovery rolls back the states of *all* machines to the latest snapshot, when processing is resumed, the computation from snapshot to the current state is repeated for machines that did not fail.

- *Increased network bandwidth usage*: The local snapshots are saved on remote machines, either on the distributed file system or in memory. All machines bulk transfer snapshots over the network simultaneously. This stresses the network and increases peak bandwidth usage. The problem gets worse in case of a multi-tenant cluster.

We develop CoRAL, a highly optimized recovery technique for asynchronous graph processing that is the first *confined recovery* technique for asynchronous processing. We

observe that the correctness of graph computations using asynchronous processing rests on enforcing the *Progressive Reads Semantics* (PR-Semantics) which results in the graph computation being in PR-Consistent State. Therefore recovery need not roll back graph state to a globally consistent state; it is sufficient to restore state to a PR-Consistent state. We leverage this observation in two significant ways. First, non-failing machines do not rollback their graph state, instead they carry out confined recovery by reconstructing the graph states of failed machines such that the computation is brought into a PR-Consistent state. Second, globally consistent snapshots are no longer required, instead *locally consistent snapshots* are used to enable recovery.

We show that our technique recovers from failures and finishes processing 1.5× to 3.2× faster compared to the traditional asynchronous checkpointing and recovery mechanism when failures impact 1 to 6 machines of a 16 machine cluster. Moreover, capturing locally consistent snapshots significantly reduces intermittent high peak bandwidth usage required to save the snapshots – the average reduction in 99th percentile bandwidth ranges from 22% to 51% while 1 to 6 snapshot replicas are being maintained.

1.1.2 Processing Dynamic Graphs

We now describe the techniques developed to accelerate processing of dynamic graphs. Dynamic graphs model relationships with temporal properties. Hence, they allow the graph structure to change over time by capturing structural mutations in form of addition and deletion of vertices and edges. These structural mutations lead to change in the vertex values which can be quickly computed using *incremental processing*, i.e., by reusing the previous results to reduce redundant computations. While incremental process-

ing accelerates overall execution, we need to ensure that graph algorithm are amenable to incremental processing so that the incremental computation converges to correct results.

We first develop optimizations to amortize the computation and communication costs while processing *evolving graphs* and then develop a dynamic dependence based incremental processing technique to quickly process *streaming graphs*.

Evolving Graph Processing

There is a great deal of interest in carrying out graph analytics tasks over evolving graphs, i.e. repeatedly computing the results of analysis at various points in time to study how different characteristics of vertices (e.g., their PageRank, shortest path, widest path, community, etc.) change over time. The analysis of an evolving graph is expressed as the *repeating* of graph analysis over multiple snapshots of a changing graph – different snapshots are analyzed independently of each other and their results are finally aggregated.

Due to the fast-changing nature of a modern evolving graph, the graph often has a large number of snapshots; analyzing one snapshot at a time can be extremely slow even when done in parallel, especially when these snapshots are large graphs themselves. For instance, one single snapshot of the Twitter graph [17] has over 1 billion edges, and there are in all 25.5 billion edges in all its snapshots we analyzed.

We develop temporal execution techniques that significantly improve the performance of evolving graph analysis, based on an important observation that different snapshots of a graph often have large overlap of vertices and edges. By laying out the evolving graph in a manner such that this temporal overlap is exposed, we identify two key optimizations that aid the overall processing:

- We reorder the computations based on loop transformation techniques to amortize the cost of fetch across multiple snapshots while processing the evolving graphs. In particular, we develop *Fetch Amortization* that *simultaneously processes* Δ snapshots $(G_{i+1}, G_{i+2} \ldots G_{i+\Delta})$ so that fetches of vertices common among some of the Δ snapshots can be *aggregated* to amortize fetch cost across snapshots.

- We enable feeding of values computed by earlier snapshots into later snapshots to amortize the cost of processing vertices across multiple snapshots. In particular, we develop *Processing Amortization* which works as follows: while snapshot G_{i-1} is being processed, when processing of G_i is initiated for simultaneous processing, processing amortization *feeds* current vertex values from snapshot G_{i-1} into G_i as initializations to accelerate the processing of G_i.

Furthermore, the two optimizations are orthogonal i.e., they amortize different costs, and hence, we identify and exploit the synergy between them by allowing feeding of values from all vertices, including those that haven't attained their final values, to amortize the processing cost, while simultaneously reordering computations to amortize the fetch cost. We demonstrate the effectiveness of these optimizations by incorporating them in GraphLab and ASPIRE. Our experiments with multiple real evolving graphs and algorithms show that, on average fetch amortization speeds up execution of GraphLab and ASPIRE by 5.2× and 4.1× respectively. Amortizing the processing cost yields additional average speedups of 2× and 7.9× respectively.

Streaming Graph Processing

The main idea behind streaming graph processing systems is to interleave iterative processing of the graph with the application of batches of graph updates. The iterative processing maintains an *intermediate approximate result* (intermediate for short) of the computation on the *most recent* version of the graph. When a query arrives, the accurate result for the *current version* of the graph where all batched updates have been applied is obtained by performing the iterative computation starting at the intermediate results, hence leveraging *incremental computation* to achieve efficiency. The intuition behind it is straightforward: the values right before the updates are a better (closer) approximation of the actual results than the initial vertex values and, hence, it is quicker to reach convergence if the computation starts at the approximate values.

However, the above intuition has an implicit assumption that is often overlooked: an intermediate value of a vertex is indeed closer to the actual result than the initial value *even when the graph mutates*. We observe that this assumption always holds for strictly growing graphs if the graph algorithm performs *monotonic computation* (e.g., SSSP, BFS, Clique, etc.), because adding new edges preserves the existing graph structure on which intermediate values were computed. However, if graph is mutated via edge deletions, the graph structure changes may *break monotonicity* and invalidate the intermediate values being maintained.

Depending on the nature of the monotonic graph algorithms, these invalid intermediate values can impact the correctness of final results and degrade the performance of incremental processing. To illustrate, consider a path discovery algorithm where the inter-

mediate path computed before the deletion (i.e., intermediate result) no longer exists and the new path to be discovered (i.e., final result) is "worse" than the intermediate path. If the algorithm only updates the vertex value (i.e., path discovered) when a new "better" path is found, the algorithm will stabilize at the non-existent old path and never converge to the correct result. On the contrary, consider a self-healing monotonic algorithm where the computation can go "backward" after an edge deletion to stabilize at the correct result. In this case, starting the computation at the intermediate result may not always be profitable. It could have taken a much less effort to reach the correct result had the computation started at the initial value after edge deletion.

To address the above issues, we present a novel runtime technique called KickStarter that computes a *safe* and *profitable* approximation (i.e., trimmed approximation) for a small set of vertices upon an edge deletion. KickStarter is the first technique that can achieve safety and profitability for a general class of monotonic graph algorithms, which compute vertex values by performing selections (discussed shortly). After an edge deletion, computation starting at the trimmed approximation (1) produces correct results and (2) converges at least at the same speed as that starting at the initial value.

The key idea behind KickStarter is to identify values that are (directly or transitively) impacted by edge deletions and adjust those values accordingly before they are fed to the subsequent computation. To achieve this, KickStarter characterizes the *dependences* among values being computed and tracks them actively as the computation progresses. However, tracking dependences online can be very expensive; how to perform it efficiently is a significant challenge. We overcome this challenge by making an observation on mono-

tonic algorithms. In many of these algorithms, the value of a vertex is often *selected* from one single incoming edge, that is, the vertex's update function is essentially a selection function that compares values from all of the incoming edges (using max, min, or other types of comparisons) and selects one to compute the value of the vertex. This algorithmic feature indicates that the value of a vertex only *depends on* the value of *one single* in-neighbor, resulting in simplified dependences and reduced tracking overhead.

Upon an edge deletion, this dependence information will be used first to find a small set of vertices impacted by the deleted edges. It will also be used to compute safe approximate values for these vertices. Our results using four monotonic algorithms and five large real-world graphs show that KickStarter not only produces correct results, but also accelerates existing processing algorithms such as Tornado [112] by 8.5–23.7×.

1.1.3 Out-of-core Graph Processing

Beyond leveraging asynchrony for distributed graph processing, we further demonstrate its efficacy across different processing environments by identifying a key opportunity to improve graph processing in an *out-of-core* setting [73, 103].

Despite much effort to exploit locality in the partition design, existing systems use *static partition layouts*, which are determined before graph processing starts and remain the same throughout processing. Repeatedly loading data from disks using these static partitions creates significant I/O inefficiencies, which impacts the overall graph processing performance. Hence, we explore the idea of exploiting algorithmic asynchrony by reordering computations, using which we further design *dynamic partitions* to reduce the above I/O inefficiency in out-of-core graph systems. Dynamic partitions truly capture the dynamic

working set of the algorithm so that only those graph elements which are relevant at a given point in time are loaded from the disk. We also develop a *delay-based accumulative* programming/execution model that enables *incremental vertex computation* by expressing computation in terms of contribution increments flowing through edges, hence maximizing the potential of dynamic shards to avoid disk I/O. Our experiments with five common graph applications over six real graphs demonstrate that using dynamic shards in GraphChi [73] accelerates the overall processing by up to 2.8×.

1.2 Dissertation Organization

The rest of the dissertation is organized as follows. Chapter 2 presents the *relaxed memory consistency model* and *cache consistency protocol* to tolerate communication latencies in distributed processing. Chapter 3 describes a *confined recovery* strategy to quickly recover from machine failures and further develops a bandwidth-sensitive *locally consistent checkpointing* strategy. Chapter 4 presents temporal execution techniques to reduce computation and communication involved in evolving graph analysis. Chapter 5 discusses a dynamic dependence based incremental processing technique to process streaming graphs. Chapter 6 demonstrates the effectiveness of asynchrony in out-of-core graph processing. Chapter 7 discusses various research works in the literature and Chapter 8 concludes the thesis as well as discusses directions for future work.

Chapter 2

Relaxed Consistency Model

Various iterative graph algorithms (e.g., graph mining, graph analytics, PDE solvers) can be expressed via asynchronous parallelism by relaxing certain read-after-write data dependences. This allows the threads to perform computations using stale (i.e., not the most recent) values of data objects. In a distributed environment, the graph is partitioned across a set of machines (nodes) and each machine operates on the subgraph residing on that machine. Hence, the access latency experienced by threads in the system varies based upon the machine at which the object is physically stored. On our *Tardis* cluster which has 16 compute nodes, the latency of accessing an object increases by 2.3× if instead of being found in the local software cache it has to be brought from a remote machine. Therefore the key to obtaining high-performance on a cluster is successfully tolerating the high inter-machine communication latency.

In this chapter, we exploit the asynchronous nature of algorithms to tolerate the long inter-machine communication latency. In particular, we leverage the availability of mul-

tiple legal executions due to inherent relaxable read-write dependences in order to quickly use the locally available values as described next.

When a thread's computation requires an object whose stale value is present in the local cache, instead of waiting for its most recent copy to be fetched from a remote machine, the stale value can be used. By tracking the *degree of staleness* of cached objects in terms of the number of object updates that have since been performed, we can limit the extent to which use of stale values is permitted. However, fruitfully exploiting this idea is challenging due to the following:

- *Allowing high degree of staleness* can lead to excessive use of stale values which in turn can slow down the algorithm's convergence, i.e. significantly more iterations may be needed to converge in comparison to the synchronous version of the algorithm. This can wipe out the benefits of exploiting asynchrony.

- *Allowing low degree of staleness* limits the extent to which long latency fetch operations can be tolerated as the situation in which the cached value is too stale to be used becomes more frequent causing delays due to required inter-machine fetches.

We address the above challenge by designing a relaxed memory consistency model and cache consistency protocol that simultaneously *maximize the avoidance of long latency communication operations* and *minimize the adverse impact of stale values on convergence.* This is achieved by the following two ideas:

- First, we use a consistency model that supports bounded staleness and we set the permissible bound for using stale values to a high threshold. By doing so, we *increase*

the likelihood of avoiding long latency fetch operations because if a stale value is present in the cache it is highly likely to be used.

- Second, we design a cache consistency protocol that incorporates a policy that *refreshes* stale values in the cache such that when a stale value is used to avoid a long latency operation, the likelihood of the value being minimally stale is increased. Thus, this refresh policy *avoids slowing down the convergence* of the algorithm due to use of excessively stale values.

Hence, significant performance improvements can be obtained by effectively tolerating inter-machine fetch latency.

The focus of our work is on asynchronous graph algorithms and we rely upon an object-based Distributed Shared Memory (DSM) to provide better programmability. In this chapter, we study a vertex centric approach for programming asynchronous algorithms with ease using DSM that is based upon a memory consistency model that incorporates bounded staleness. We design a consistency protocol that tracks staleness and incorporates a policy for refreshing stale values to tolerate long latency of communication without adversely impacting the convergence of the algorithm. Our experiments show that the use of this fetch policy is critical for the high performance achieved. Finally, we demonstrate that, on an average, our asynchronous versions of several graph algorithms outperform algorithms based upon: prior relaxed memory models that allow stale values by at least 2.27x and Bulk Synchronous Parallel (BSP) [126] model by 4.2x.

The rest of the chapter is organized as follows. Section 2.1 shows how we express asynchronous parallel algorithms. In Section 2.2, we present the *relaxed object consistency*

model that incorporates the use of stale objects for fast access and the design of our *DSM system* that implements the relaxed execution model. The key component of DSM, the *relaxed consistency protocol*, is presented in Section 2.3. We discuss the implementation of our prototype, experimental setup, and results of evaluation in Section 2.4 and Section 2.5.

2.1 Asynchronous Parallelism

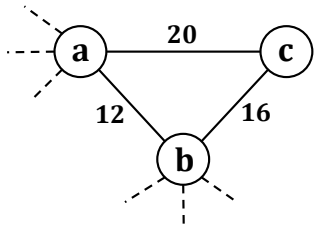
We consider various iterative algorithms that move through the solution search space until they converge to a stable solution. In every iteration, the values are computed based on those which were computed in the previous iteration. This process continues until the computed values keep on changing across subsequent iterations. The asynchronous model for parallel computation allows multiple threads to be executed in parallel using a set of possibly outdated values accessible to those threads. By using an outdated value, a thread avoids waiting for the updated value to become available. Algorithm 1 shows a basic template for such convergence based iterative algorithms. A set of threads execute the DO-WORK method which iteratively performs three tasks: fetch inputs, compute new values, and store newly computed values. At the end of DO-WORK, if a thread detects that the values have not converged, it votes for another iteration. This process ends when no thread votes for another iteration, which is detected in the MAIN method.

The DSM-FETCH method fetches an object from the DSM (line 7). For example, in a vertex centric graph algorithm, the vertex to be processed is initially fetched using this method. Then, to fetch its neighbors, again, DSM-FETCH is used (line 10). In synchronous versions of these algorithms, this DSM-FETCH method incurs very high latency because it

may result in a remote fetch to access the most recent value. However, when these algorithms are implemented using the asynchronous computational model, some of the objects used in the computation are allowed to reflect older values, i.e., they do not reflect the most recent changes. Hence, these methods may return a stale vertex value.

On a DSM, objects can be physically stored at different machines and the application remains unaware of the underlying placement of these objects. This leads to non-uniform object access time i.e., objects placed on local machine are available faster than those placed on remote machines. As expected, the task of fetching and storing data (marked in red) on a distributed environment involves network access which incurs very high overhead.

Let us consider the single source shortest path (SSSP) algorithm which computes the shortest path from a given source node to each node in a graph. To guarantee convergence, the iterative algorithm assumes that the graph does not have a negative cycle. Figure 2.1 shows an example sub-graph along with the distance values calculated for nodes a and b at the end of iterations $i = 0, 1, 2, 3$ and the initial value for c at the end of iteration $i = 0$. Since the algorithm is implemented based on asynchronous parallelism, a perfectly valid execution scenario can be as follows:



	0	1	2	3
d(a)	20	15	15	15
d(b)	28	20	10	10
d(c)	40	?	?	?

Figure 2.1: An example subgraph for SSSP.

Algorithm 1 A basic template for Iterative Algorithms

```
1: function DO-WORK(thread-id)
2:     curr ← GET-START-INDEX(thread-id)
3:     end ← GET-END-INDEX(thread-id)
4:     error ← ϵ
5:     while curr < end do
6:         oid ← GET-OBJECT-ID(curr)
7:         object ← DSM-FETCH(oid)
8:         r-objects ← ∅
9:         for r-id ∈ object.get-related-object-ids() do
10:            r-objects ← r-objects ∪ DSM-FETCH(r-id)
11:        end for
12:        old-value ← object.get-value()
13:        comp-value ← f (object, r-objects)
14:        object.set-value(comp-value)
15:        DSM-STORE(object)
16:        error = MAX(error, |old-value − comp-value|)
17:        curr ← curr + 1
18:    end while
19:    /* Local termination condition */
20:    if error > ϵ then
21:        vote to continue
22:    end if
23: end function
24:
25: function MAIN
26:    INITIALIZE-DSM(object-set)
27:    do
28:        parallel-for all threads do
29:            DO-WORK(thread-id)
30:        end parallel-for
31:        barrier
32:        /* Global termination condition */
33:    while at least one thread votes to continue
34: end function
```

- During $i = 1$, c observes $d^0(a) = 20$ and $d^0(b) = 28$. Hence, $d(c)$ is set to $min(20 + 20, 16 + 28) = 40$.

- During $i = 2$, only updated value $d^1(b) = 20$ is observed by c and $d(c)$ is set to $min(20 + 20, 16 + 20) = 36$.

- During $i = 3$, c observes that $d^2(a) = 15$, but it remains oblivious to the change in $d(b)$ which leads to $d(c)$ to be set to $min(20 + 15, 16 + 20) = 35$.

- During $i = 4$, c observes that $d^2(b) = 10$ and that $d^2(a)$ is still the same. Hence, $d(c)$ is set to $min(20 + 15, 16 + 10) = 26$.

In the above execution, $d(c)$ was always computed using $d(a)$ and $d(b)$, one of which did not reflect recent changes. Using older values made the intermediate values of $d(c)$ inconsistent. However, if the updated values of both $d(a)$ and $d(b)$ are guaranteed to be observed during future iterations, the algorithm converges to its correct solution. Hence, an intuitive and straightforward way to hide fetch latencies in these algorithms is to allow the use of previously fetched older values. This can be achieved by using *delta coherence* as shown in [21], that allows objects which are no more than x versions out-of-date. However, statically maintaining x as a threshold will not yield the expected performance benefits.

Figure 2.2 shows execution instances when staleness threshold is statically set to 0, 1 and 2. When the threshold is 0, change in $d(b)$ from 28 to 22 at the end of iteration 1 is immediately seen in iteration 2. This change gets hidden when the threshold > 0; for e.g., with threshold = 1, $d(b) = 21$ is noticed directly in iteration 3 and $d(b) = 22$ is never seen. As we can see in Figure 2.2, setting the staleness threshold to 1 or 2 allows computations to

	0	1	2	3	4	5	6	7	8	
d(a)	20	-	16	-	-	15	14	13	-	
d(b)	28	22	21	-	-	18	17	9	-	
d(c)	40	-	38	36	-	-	34	33	25	*threshold = 0*
d(c)	40	-	-	37	-	-	35	33	-	*threshold = 1*
d(c)	40	-	-	-	-	-	34	-	-	*threshold = 2*

Figure 2.2: Execution instances showing intermediate values of $d(c)$ for statically set staleness thresholds of 0, 1 and 2.

avoid immediate fetches; however, the computations choose to work with stale values (for e.g., in iterations 4 and 5) even when algorithms could have progressed using fresher values. These computations can be considered redundant or wasteful. Note that $d(a)$ and $d(b)$ can continue to remain unchanged (as in iterations 3 and 4) across a long series of consecutive iterations, making the situation worse for any threshold > 0. A key observation is that if any subset of the set of values is used to compute the new value updates across subsequent iterations, the algorithm can be deemed to have progressed across these iterations. Hence, it is important for the updated values to be observed by required computations in order to make a faster progress towards the correct stable solution.

Also, when a requested object's staleness has already crossed the static threshold, a DSM fetch is required. This enforces a limit on the number of remote fetches that can be avoided. For example, for a threshold x, every x^{th} access to an object can potentially cause a fetch from its global copy which may be present on a remote location.

Note that SSSP's monotonic nature along with the miniature subgraph in the example allows the discussion at hand to be simple; however, the observations drawn from this example apply to other more complex situations too.

To formalize the discussion, we define following terms:

- **Current Object** is one whose value reflects the most recent change.
- **Stale Object** is one which was current at some point in time before the present time.
- **Staleness of an object** is the number of recent changes which are not reflected in the object's value.

In our example, during $i = 2$, $d^1(a) = 20$ is a current object and $d^0(b) = 28$ is a stale object. It is easy to follow that the staleness value of current objects is always 0.

In summary, we draw the following conclusions for asynchronous versions of convergence based iterative algorithms.

- Since these algorithms do not enforce strict data dependence constraints (in particular, read-after-write dependences for objects), they can tolerate use of stale objects.
- To maintain a good convergence rate, it is recommended that these algorithms rely more on the current values of objects and less on the stale values of objects.

Even though these conclusions inherently seem contradictory, they give us a key insight that maintaining a right mix of current and stale objects along with carefully balancing staleness of these objects can lead to better performance of these algorithms on DSM.

By allowing computations to use stale objects, the path to the final solution in the solution search space may change. This means that the total number of iterations required to converge to the final solution may also vary. However, since local caches can quickly provide stale objects, data access will be faster and time taken to execute a single iteration will drastically reduce. This reduction in time can result in significant speedups for various iterative algorithms if we minimize the staleness of values available without stalling computation threads. Note that since stale objects are often used for computations, termination of these iterative algorithms needs to be handled carefully and additional checks must be performed along with the algorithm specific termination conditions (discussed further in Section 2.3).

This analysis motivates the need for a relaxed model that can provide fast access to, possibly old, data that is minimally stale in order to achieve better performance for convergence based iterative algorithms.

2.2 Relaxed Object Consistency Model

The relaxed object consistency model we present accomplishes two goals. First, it achieves *programmability* by providing a single writer model that makes it easy to reason about programs and intuitive to write correct asynchronous parallel algorithms. Second, it enables *high performance* through low latency access of objects which requires careful (minimal) use of stale objects. To achieve the above goals, we have identified four constraints that together describe our consistency model and are enforced by our cache consistency protocol. Next we present these constraints and an overview of how they are enforced.

Object consistency constraints *for Programmability.* We define our consistency model with respect to a *single object*, i.e. we do not enforce any ordering among operations on different objects even if they are potentially causally related. For each object, we rely upon having a *single writer*, i.e. the same machine is responsible for updating a particular data item in every iteration. Our iterative object centric approach for programming asynchronous algorithms naturally maps to the single writer discipline and allows the programmer to intuitively reason about the program. We enforce the single writer discipline by fixing the assignment of computations to machines such that threads on the same machine update the same set of objects in every iteration. Although our consistency model does not enforce any ordering on the writes to an object from different machines, the programmer does not need to be concerned about this as chaotic writes to the same object by multiple machines are prohibited by ensuring that there is only a single writer for each object. Using the single writer discipline gives us another advantage – our consistency protocol does not have to deal with multiple writes to same objects. This simplifies the consistency protocol by eliminating the need to maintain write/exclusive object states. Now we are ready to state two of our constraints on writes and reads to each object and describe their enforcement.

(Local Updates) *Local writes must be immediately visible.* This constraint enforces an ordering on multiple writes to an object by the same machine. To satisfy this constraint and provide strict ordering of writes to an object by its single writer, threads in our system do not maintain any thread-local cache and all writes directly go to the global copy of the object. Our system employs *machine level caches* to make remote objects locally available; these caches are *write through* to make writes visible across different machines.

(Progressive Reads) *Once an object is read by a thread, no earlier writes to it can be read by the same thread.* This constraint makes our model intuitive to programmers by guaranteeing that the updated values for an object will be seen in the order of its writes. Since we only have one global copy of an object at its home machine, any stale-miss or a refresh on stale-hit (described later) at another machine will make its local copy current.

As we see, the above two constraints are primarily required to guarantee correctness and allow programmers to intuitively reason about program execution.

Object consistency constraints *for Performance.* For high performance we must permit the use of stale objects and avoid long latency communication operations. The constraints we present next involve the use of stale object values.

(Bounded Staleness) *A read is guaranteed to receive an object whose staleness is no more than a particular threshold.* A bound on staleness allows the threads to notice the updated values at some point in the future. This constraint is satisfied by altering the definition of a cache hit as described in the next section. The strictness of this bound can be relaxed using asynchronous invalidate messages, as done in our protocol.

(Best Effort Refresh) *A series of reads by the same thread for the same object should preferably reflect updated values, independent of the threshold.* The previous constraint alone does not guarantee that updates will be observed by threads that depend on those updates. Hence, this final constraint is introduced to allow threads to quickly observe the updated values which helps the algorithm to progress at a faster rate. This final constraint is enforced by our cache consistency protocol which specifically employs a mechanism for *asynchronously refreshing objects on stale-hits* to allow fast access to updated objects.

Any DSM implementation that wishes to satisfy our object consistency model *must* satisfy the first three constraints. This does not mean that the fourth constraint can be ignored. Even though the fourth constraint is a loose constraint, the protocol is expected to do its best to satisfy this constraint.

2.3 Relaxed Consistency Protocol

Next we introduce the new consistency protocol which satisfies the model proposed in the previous section. In Section 2.3.1, we introduce various notations and terms which will be used to discuss the working of the protocol in Section 2.3.2.

2.3.1 Definitions and Notation

Formally, $M = \{m_0, m_1, \ldots, m_{k-1}\}$ is the set of k machines (nodes) in the cluster. The mapping function h maps an object o to its *home machine* m_i i.e., on DSM, if o resides on m_i, then $h(o) = m_i$.

Every machine $m_i \in M$ has a cache c_i which locally stores objects and tracks their staleness. An entry in the cache is of the form $\langle o, staleness \rangle$ where o is the actual object and *staleness* is its staleness value. Since we do not use thread-level caching, these caches provide the fastest data access.

Every machine $m_i \in M$ has a directory d_i to track the set of machines which access the objects mapped to that machine. A directory entry for an object o is of the form $d_i^o = \{m_j \mid m_j \in M \text{ and } o \in c_j\}$ $\forall$ o such that $h(o) = m_i$.

Also, we keep a threshold t which is used to determine the usability of locally available o. Hence, o can be considered

$$\begin{aligned} current &\quad \text{if } staleness = 0 \\ stale &\quad \text{if } 0 < staleness \leq t \\ invalid &\quad \text{if } staleness > t \end{aligned}$$

We change the meaning of a hit and a miss in the cache as follows. If the requested object in local cache is either current or stale, it is a hit. Otherwise, it is a miss. Hence, for an object o requested at a machine m_i, we determine a hit or a miss as follows:

$$\begin{aligned} hit &\quad \text{if } o \in c_i \text{ and } staleness \leq t \\ miss &\quad \text{otherwise} \end{aligned}$$

For ease of discussion, we further categorize a hit and a miss. For a requested object which is present in the local cache, it is a *current-hit*, a *stale-hit* or a *stale-miss* if the object in cache is current, stale or invalid, respectively. If the requested object is not in the local cache, it is simply a *cache-miss*. Hence, for an object o requested at a machine m_i, the result can be one of the following:

$$\begin{aligned} current\text{-}hit &\quad \text{if } o \in c_i \text{ and } staleness = 0; \\ stale\text{-}hit &\quad \text{if } o \in c_i \text{ and } 0 < staleness \leq t; \\ stale\text{-}miss &\quad \text{if } o \in c_i \text{ and } staleness > t; \\ cache\text{-}miss &\quad \text{if } o \notin c_i. \end{aligned}$$

2.3.2 Protocol

In this section, we discuss the basic working of our protocol using the terms introduced in the previous section. The traditional directory based coherence mechanism [76, 18] is useful to enforce strict consistency. Our protocol extends the directory based protocol to control the degree of coherence, as required at runtime. In the following discussion, we assume that machine m_i requests for object o.

On a cache-miss, m_i first sends a read request to $m_j = h(o)$. On receiving the read request from m_i, m_j sets $d_j^o \leftarrow d_j^o \cup \{m_i\}$. After fetching o from the DSM, m_i adds $\langle o, 0\rangle$ to c_i . While adding $\langle o, 0\rangle$ to c_i, if c_i is full, object o' is evicted from c_i based on the Least Recently Used (LRU) replacement policy. To evict o', m_i sends an eviction message to $m_p = h(o')$. On receiving the eviction message from m_i, m_p sets $d_p^{o'} \leftarrow d_p^{o'} \setminus \{m_i\}$.

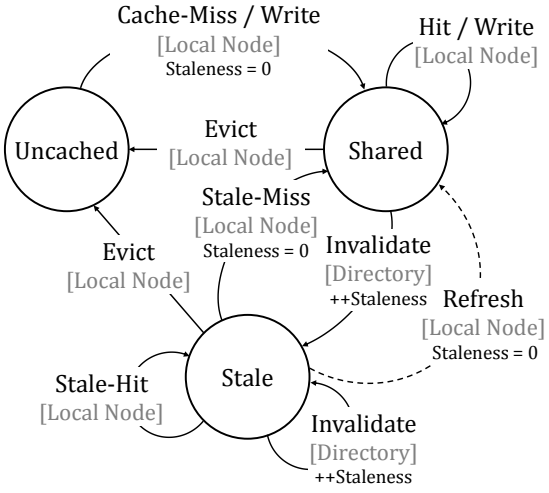
When m_i writes o back to the DSM, it sends a write message to $m_j = h(o)$ and continues immediately. On receiving the write message from m_i, m_j *asynchronously* sends an invalidation message to all $m_q \in d_j^o \setminus \{m_i\}$ and sets $d_j^o \leftarrow d_j^o \cup \{m_i\}$. When m_q receives invalidation for o, it sets $\langle o, staleness\rangle \leftarrow \langle o, staleness + 1\rangle$ in c_q. We use invalidate-on-writes instead of broadcasting updates so that we can avoid consecutive updates to be propagated to remote nodes which makes the intermediate updates, before the object is actually read, redundant.

On a stale-miss, the current value of o is fetched from the DSM and m_i sets $\langle o, staleness\rangle \leftarrow \langle o_{curr}, 0\rangle$ in c_i.

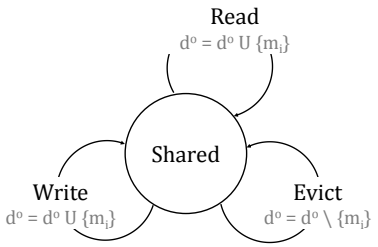
On a current-hit, the local copy of o is used by m_i. No further processing is required in this case.

On a stale-hit, the local copy of o is used by m_i and a DSM fetch request is issued asynchronously to *refresh* the local copy of o. When the current value of object is received from the DSM, m_i sets $\langle o, staleness \rangle \leftarrow \langle o_{curr}, 0 \rangle$ in c_i.

By allowing cache misses in the traditional protocol to be considered as cache hits in our protocol, $o \in c_i$ can remain outdated until its $staleness \leq t$. To allow visibility of more recent values of o for subsequent reads on m_i, the protocol incorporates *refresher threads*. The refresher thread observes that m_i has read a stale value of o from c_i as its $staleness > 0$; hence, it initiates a fetch to update $o \in c_i$ with its current value from the DSM. This prevents o from remaining outdated for long time and thus causes subsequent reads to receive fresher values.



(a) State transition diagram for cache entries. The operations are shown in black on the transition and the source of those operations are shown in gray.



(b) State transition diagram for directory entries. The protocol messages are shown in black on the transitions and the operations to maintain the set of machines currently accessing the object are shown in gray.

Figure 2.3

Figure 2.3a shows the state transition diagram for object entries in machine caches. The gray text in parentheses indicates the source of the event and the small text at the bottom of the transitions show how an object's staleness is maintained. The shared state represents that the object is current. On receiving an invalidation message for a current object, the state of object changes to stale state. Every invalidation message increments the staleness by 1. A hit occurs when the object is either in shared or stale state and the staleness of the object is at most equal to the threshold value. If the current staleness is greater than the threshold value, a stale-miss occurs and the current value is fetched. This allows the state of the object to be changed to shared state. Figure 2.3b shows the state transition diagram for object entries in directories. The gray text on transitions indicates how the set of machines locally accessing the object is maintained. Since the global copies in DSM are always current, the object is always considered to be in shared state. The set of machines having copies of the object (stale or current) is maintained appropriately during the incoming read, write, and evict requests. Each write request leads to invalidation messages to respective set of machines.

Termination Semantics. Since stale objects will often be used during computations, termination of iterative algorithms needs to be handled carefully. Along with the algorithm specific termination conditions, additional checks must be performed to make sure that the final values are not computed using stale values, i.e., all the required updates are visible to all machines. This is achieved by making sure that there are no outstanding invalidate or refresh messages and that all the objects used for computation in last iteration were current objects.

2.3.3 Optimizations

The above protocol satisfies the relaxed object consistency model proposed in Section 2.2. To further enhance the overall performance of the system, we perform the following standard optimizations over the protocol.

Work Collocation. Since the single writer model requires a unique node to perform all computation that updates a given object throughout the execution, the home node can itself be used as the single writer. In other words, computation for objects can be assigned to the nodes that maintain the global copy of those objects. This eliminates the write latency of remote objects and reduces the protocol traffic as write messages are converted into local signals on the home machine.

Message Aggregation. When multiple messages are sent to the same destination node, these messages can be aggregated into fewer number of messages (similar to bulk transfer in [80]). Message aggregation can significantly reduce the latencies incurred by transfer of multiple small messages since individually sending small messages is significantly slower than sending fewer number of large messages.

Message Flattening. When same requests are made for the same destination node, messages can be flattened to remove redundant requests and send minimal messages to the destination node. Message flattening becomes useful when multiple computations depend on a remote high degree vertex (flattening read and refresh messages) or when same objects are updated multiple times (flattening invalidation messages).

Object replication. If the nature of computation is known prior to execution, the objects required for computations on a given node can be replicated in the machine caches during startup to reduce remote access requests for warming up the caches in the the first iteration.

2.4 Experimental Setup

In this section, we discuss few details of our system, the benchmark programs and the inputs used to evaluate our relaxed consistency protocol.

2.4.1 System Prototype

Next we describe our prototype implementation including the design of the DSM, runtime, and the cluster it runs on.

DSM. To avoid the complexities introduced by false sharing and coherence granularity, we built an object based DSM in C++ similar to *dyDSM* [66]. The objects are distributed across the cluster such that there is exactly one global copy of each object in the DSM. METIS [61] is used to partition graphs across various machines to minimize edge-cuts. The relaxed consistency protocol was implemented in the DSM to relax the strict consistency and leverage stale values. Each node maintains a directory which is populated during initialization based on the objects placed on that node. Also, each node maintains a fixed size local cache for faster object access. The size of the local cache is large enough to hold objects required for computations on the local machine. However, we do not replicate all objects in these caches during initialization and allow them to fill up only with objects that are needed on that node.

The protocol is implemented via a set of threads on each node where each thread is responsible for a separate function. These threads primarily communicate through the read, write, evict, and invalidation messages and perform required operations to maintain the directory and cache metadata. The protocol messages are communicated using MPI send/recv commands. Message aggregation is used to combine multiple messages for the same destination in order to reduce communication latencies.

The heart of our protocol is the way it satisfies the *Best Effort Refresh* constraint enforced by our model. A separate *refresher thread* is responsible for updating the objects that will be required in the near future. The refresher thread blocks on a *refresh-queue* which maintains *object-ids* of objects that need to be refreshed. It picks up the object-id from this refresh-queue and issues a fetch from dyDSM. After receiving the current object, the refresher thread updates the stale copy in local cache with its current value.

Runtime. To evaluate the effectiveness of our protocol, we have developed a runtime that facilitates writing of parallel algorithms that run using the DSM. Each node has a single computation thread which repetitively executes the DO-WORK method as presented in Algorithm 1. The workload is distributed across all the nodes during initialization and later, the computation thread works on the same workload in every iteration. Hence, this implementation satisfies the single writer model. Since the computation thread is expected to block when the required protocol read, write, and evict requests are sent to the home machine, sending of these messages is taken care by the computation thread itself.

The computation thread is responsible for communicating the need to refresh objects which it will require in near future. On a stale-hit, the computation thread uses the

Application	Type
Heat Simulation (HS) Wave Simulation (WS)	Partial Differential Equations (PDEs)
Graph Coloring (GC) Connected Components (CC) Community Detection (CD) [83] Number of Paths (NP)	Graph Mining
PageRank (PR) [94] Single Source Shortest Path (SSSP)	Graph Analytics

Table 2.1: Convergence based Iterative Algorithms.

locally available stale object. However, before using this stale object, it enqueues the object-id in the refresh-queue. Since the refresh thread will update the object with its current value in the local cache, the next request for this object by the computation thread will reflect its refreshed value, allowing computations to observe updated values. Staleness of objects used for computation is checked to ensure that termination is done correctly. Also, message queues are checked to be empty to make sure that there are no outstanding messages.

To transfer objects to and from the DSM, the runtime provides DSM-Fetch and DSM-Store methods. This hides the internal details of the protocol and allows parallel algorithms to directly execute over the DSM using these two methods.

System. We evaluate our protocol on *Tardis* which is a commercial 16-node cluster, running CentOS 6.3, Kernel v2.6.32-279. Each node has 64 GB memory and is connected to a Mellanox 18 port InfiniBand switch.

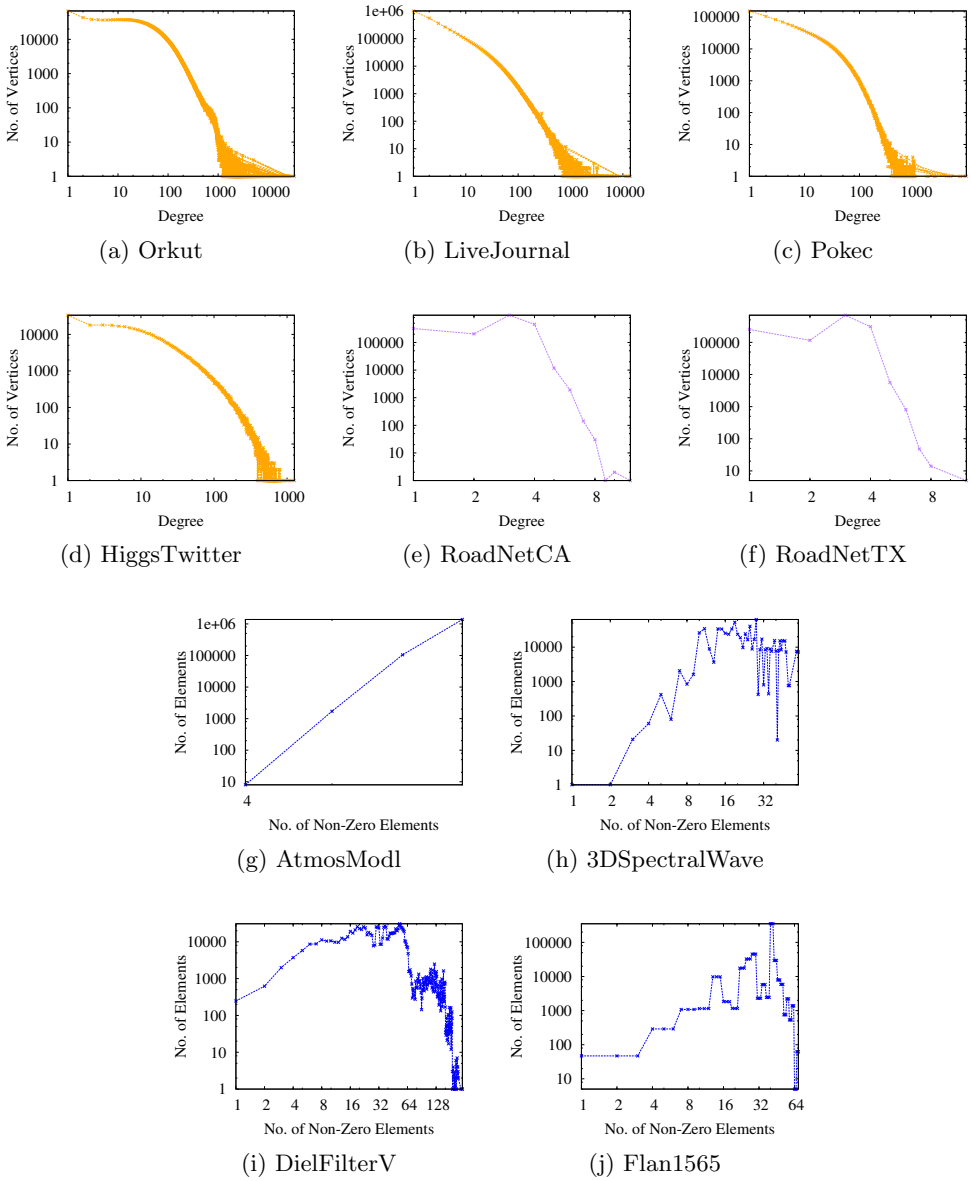


Figure 2.4: Real-world graph datasets used for evaluation of graph mining and analytics benchmarks and sparse matrices used for PDE benchmarks.

2.4.2 Benchmarks and Inputs

We use a wide range of modern applications (as listed in Table 2.1) and evaluate their asynchronous implementations. These applications are based on vertex centric model

Graph	Edges	Vertices
Orkut [138]	234,370,166	3,072,441
LiveJournal [138]	68,993,773	4,847,571
Pokec [121]	30,622,564	1,632,803
HiggsTwitter [29]	14,855,875	456,631
RoadNetCA [78]	5,533,214	1,971,281
RoadNetTX [78]	3,843,320	1,379,917
AtmosModl [28]	10,319,760	1,489,752
3DSpectralWave [28]	30,290,827	680,943
DielFilterV3Real [28]	89,306,020	1,102,824
Flan1565 [28]	114,165,372	1,564,794

Table 2.2: Real-world graphs & matrices used in experiments.

where each vertex iteratively computes a value (e.g., colors for GC, ranks for PR, and shortest paths for SSSP) and the algorithm stops when these vertex values become stable. We obtained these algorithms from various sources, implemented them in C++, and inserted the DSM fetch and store calls. Because of their vertex centric nature, they follow the same template as shown in Algorithm 1. These applications belong to important domains (e.g., scientific simulation and social network analysis) and are divided into following three categories.

(i) Partial Differential Equations. The algorithms for solving partial differential equations (PDEs) are convergence based iterative algorithms which makes them suitable for asynchronous parallelism. We implemented two benchmarks which solve specific PDEs, namely, heat simulation (HS) and wave simulation (WS). Both the benchmarks iteratively determine the value of current cell based on the values of neighboring cells. The algorithms converge when all the cell values stabilize based on a pre-specified tolerance.

(ii) Graph Mining. These set of applications analyze various structural properties of graphs. We implemented four applications in this category: Graph Coloring (GC), Connected Components (CC), Community Detection (CD), and Number of Paths (NP). CC and CD are based on iterative label propagation [152]. The vertices are assigned labels which are initialized during setup. In every iteration, the label of a vertex is chosen based on its current label and the labels of its neighboring vertices. For CC and CD, the vertices start with unique labels; however, subsequent iterations in CC choose the minimum label whereas those in CD choose the most frequent labels [83]. For GC, the vertices are initialized with an invalid color and in subsequent iterations, a vertex is assigned a unique color which is not assigned to any of its neighbors. If two neighboring vertices are assigned the same color, one of the vertex (chosen arbitrarily but fixed) is assigned a new unique color. For NP, all vertices except the source vertex have number of paths initialized to 0 and for source vertex, it is initialized to 1. In subsequent iterations, vertices calculate the number of paths by adding those of their neighbors.

(iii) Graph Analytics. These applications model their problems as a graph and are targeted to address specific queries which are not dependent on the structure of graphs alone. PageRank (PR) and Single Source Shortest Path (SSSP) fall in this category. For SSSP, all vertices except the source vertex have their distance initialized to ∞ and for source vertex, it is initialized to 0. In subsequent iterations, distance to a vertex is calculated by summing up the distances of all neighboring vertices and the weights on corresponding edges connecting those vertices and then, choosing the minimum sum. PR is a popular algorithm which iteratively computes the rank of a page based on the ranks of its neighbors [94].

Input Data Sets. We ran the benchmarks on publicly available [77, 28] real-world graphs and matrices listed in Table 4.8. The graph inputs are used to evaluate the Graph Mining and Analytics benchmarks, whereas the matrices are used for PDEs.

		Orkut	LiveJournal	Pokec	HiggsTwitter	RNCA	RNTX
CD	**SCP**	1,530	1,141	572	70	2.64	1.10
	RCP	974	307	238	36	1.95	1.17
CC	**SCP**	1,846	1,045	316	261	77	62
	RCP	710	316	154	78	69	66
GC	**SCP**	1,568	629	228	72	0.53	0.56
	RCP	733	254	101	35	0.95	0.64
NP	**SCP**	182	141	81	31	139	174
	RCP	124	117	39	12	140	179
PR	**SCP**	4,191	3,754	1,767	602	12	7.19
	RCP	2,710	2,047	275	88	11	8.39
SSSP	**SCP**	1,735	759	248	49	74	71
	RCP	714	317	118	39	72	73

Table 2.3: Execution times (in sec) of SCP and RCP for various graph mining and analytics benchmarks on a 16-node cluster.

The graphs cover a broad range of sizes and sparsity (as shown in Fig. 2.4) and come from different real-world origins. *Orkut*, *LiveJournal* and *Pokec* are directed social networks which represent friendship among the users. *HiggsTwitter* is a social relationship graph among twitter users involved in tweeting about the discovery of Higgs particle. *RoadNetCA (RNCA)* and *RoadNetTX (RNTX)* are the California and Texas road networks respectively, in which the roads are represented by edges and the vertices represent the intersections. *AtmosModl*, *3DSpectralWave*, *DielFilterV3Real* and *Flan1565* are sparse matrices (as shown in Fig. 2.4) which represent models from various domains like atmospheric models, electromagnetics, hexahedral finite elements, and 3D consolidation problem. The graph inputs are used to evaluate the Graph Mining and Analytics benchmarks, whereas the matrices are used for PDEs.

2.5 Experimental Results

Now we present a detailed evaluation of our system including comparison with closely related protocols and systems.

2.5.1 Benefits of Exploiting Staleness

To study the benefit of using stale values we compare the performance of the following two protocols:

- **RCP**: This is the *Relaxed Consistency Protocol* developed in this work. The threshold is set to a very high number [1] (=100) and refresher threads are used; and
- **SCP**: This is the *Strict Consistency Protocol* that does not allow the use of stale values at all and is based upon the traditional directory-based write through cache coherence strategy. This is used as the baseline, to evaluate the above protocols that allow the use of stale values.

In order to better understand the effectiveness of our protocol, we do not use the *object replication* optimization during this evaluation.

Across inputs. Table 2.3 and Table 2.4 compare the execution times (in sec) for *SCP* and *RCP* on a 16-node cluster. On an average, *RCP* achieves 4.6x speedup over *SCP* for PDE benchmarks and 2.04x speedup for graph mining and analytics benchmarks. The speedups vary across different benchmark and input combinations: for example, speedups for PR

[1] Since our RCP protocol does a good job in satisfying the Best Effort Refresh constraint, the need of using low threshold values is eliminated. Through experiments, we found that thresholds above 4 do not show any difference mainly because the refresher threads quickly eliminate objects with higher staleness values.

		Atmos-Modl	3DSpec-tralWave	DielFilter-V3Real	Flan-1565
HS	**SCP**	110	464	75	180
	RCP	23	40	29	93
WS	**SCP**	61	237	106	218
	RCP	14	28	48	149

Table 2.4: Execution times (in sec) of SCP and RCP for PDE benchmarks on a 16-node cluster.

across different inputs range from 1.02x to 6.8x whereas for HS, they vary from 1.9x to 11.6x. Note that *RCP* and *SCP* give similar performance for *RoadNetCA* and *RoadNetTX*. This is because these graphs are sparse and do not have a skewed degree distribution (Figure 2.4e and Figure 2.4f); hence, partitioning them over the DSM leads to very few edge cuts. Thus, *SCP* does not suffer much from remote fetch latencies that are tolerated by *RCP*.

Across configurations. The speedups achieved by *RCP* over *SCP*, on clusters of different sizes, are shown in Figure 2.5. These speedups are based upon the *Pokec* graph input for graph analytics and mining algorithms and the *AtmosModl* matrix input for PDEs. On average, *RCP* achieves 1.6x speedup on 2 nodes, 1.9x speedup on 4 nodes, 2.6x on 8 nodes and 3.3x on 16 nodes. Since *RCP* mainly focuses on reducing remote fetches by using locally available stale values, speedups achieved by *RCP* increase as the cluster grows. This is also the reason for achieving no performance benefits for few benchmarks (e.g., CD, NP) when the cluster is small; the overheads of the system mask the little benefits achieved by the protocol.

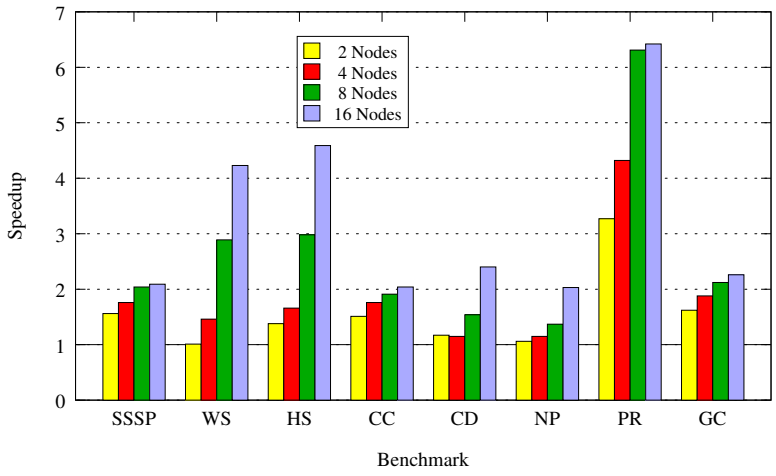


Figure 2.5: Speedups achieved by RCP over SCP on clusters of 2, 4, 8, and 16 nodes.

2.5.2 Bounded Staleness vs. RCP

The *delta coherence* [21] protocol supports *bounded staleness* by allowing use of objects which are no more than x versions out-of-date. In other words, it allows the use of stale values by statically using x as the staleness bound, but it does not use refresher threads. In this section we demonstrate that via the use of stale values delta coherence can tolerate remote access latency, but the use of stale values slows down the algorithm's convergence. In the remainder of this section, **Stale-n** refers to the delta coherence protocol with a staleness bound of n. In order to separate out the benefits achieved from latency tolerating property of *RCP*, we relax the writes in *SCP* similar to that in *RCP*. We denote *SCP* with relaxed writes as **SCP+RW**. Writes in *Stale-n* are also similarly relaxed. The detailed results presented are based upon the *Pokec* graph input for graph analytics and mining algorithms and the *AtmosModl* matrix input for PDEs.

The execution times of *RCP* and *Stale-n* ($n = 1, 2, 3$), normalized with respect to the *SCP+RW*, are shown in Figure 2.6. *RCP* consistently achieves better performance

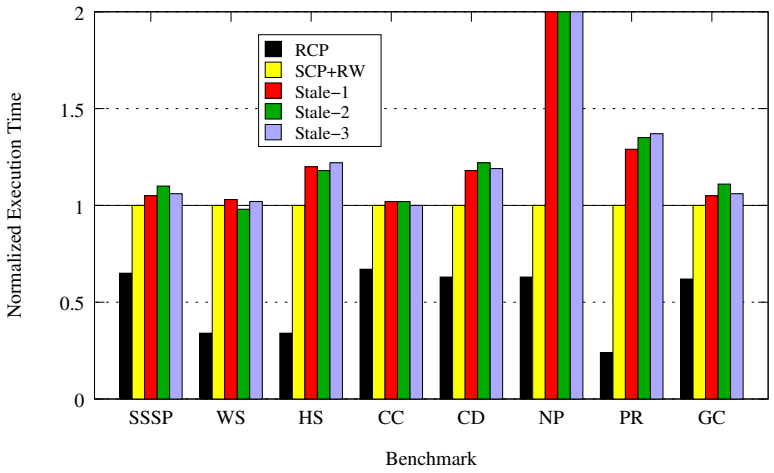


Figure 2.6: Execution times of RCP and Stale-n ($n = 1, 2, 3$) on a 16-node cluster normalized wrt SCP+RW.

than all other protocols considered. On an average, *RCP* executions times are lower than *SCP+RW* by 48.7%. The performance of *Stale-n* varies across different thresholds because when the threshold is increased, the convergence gets adversely affected (e.g., PR, SSSP). No staleness value ($n = 1, 2, 3$) consistently performs the best for *Stale-n*. On an average, the best performing *Stale-n*, which sometimes performs better than *SCP+RW* (e.g., WS), increases execution time over *SCP+RW* by 10.5% while the worst performing *Stale-n* always performs worse than *SCP+RW*. On an average, *RCP* gives 56% reduction in execution time over the best *Stale-n*.

To further analyze the relative performances of *RCP* and *Stale-n*, we present additional data, once again normalized with respect to *SCP+RW*. Let us consider the fraction of remote fetches that fall on the critical path of execution, i.e. they cause the computation thread to stall while waiting for the remote fetch to complete. From the results shown in Figure 2.7 we determine, that on an average, computation thread under *RCP* blocks for

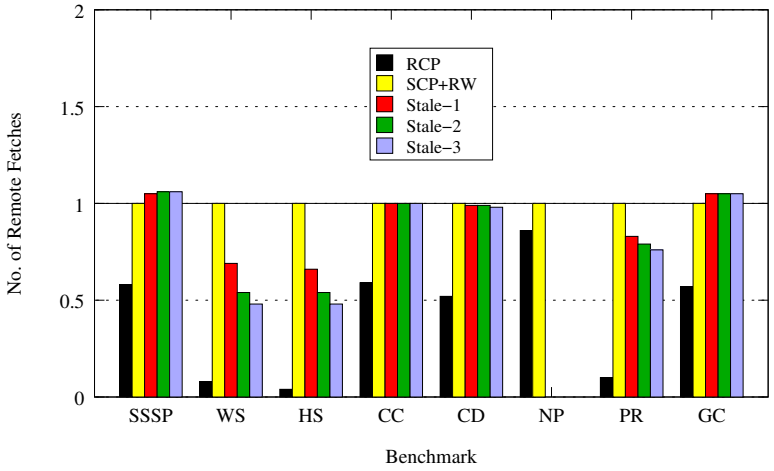


Figure 2.7: Number of remote fetches that stall computation threads normalized wrt SCP+RW.

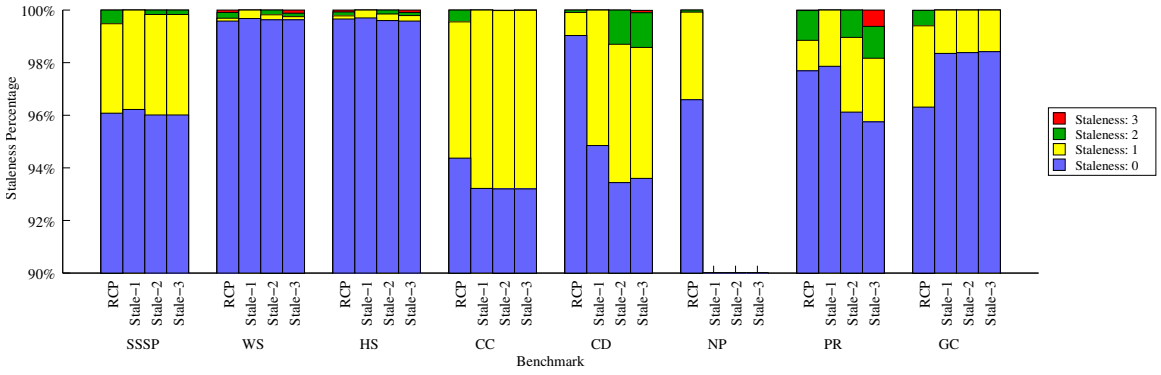


Figure 2.8: Percentage of objects used with staleness n.

only 41.83% of remote fetches which are due to compulsory cache misses. In contrast, the best *Stale-n* causes stalls on 85.6% of remote fetches. Since remote fetches are long latency operations, we expect *RCP* to perform better than both *SCP+RW* and best *Stale-n*.

Figure 2.8 shows the distribution of staleness of values used. We observe that in *RCP* the staleness of values is typically 0 or 1 – in fact on an average 97.4% of values have staleness of 0 and 2.2% of values have staleness of 1. *Stale-2* and *Stale-3* use slightly more stale values in CC, CD and PR. It is interesting to see that in GC, *RCP* uses more stale values than *Stale-2* and *Stale-3*; this is mainly because *Stale-2* and *Stale-3* use stale

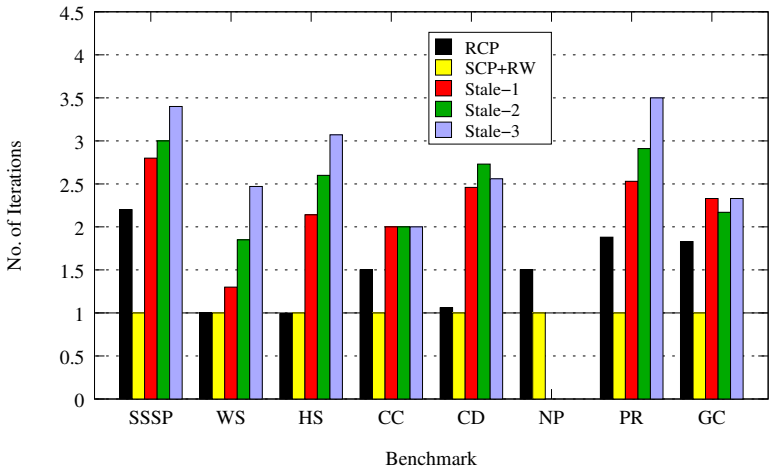


Figure 2.9: Number of iterations performed before converging normalized wrt SCP+RW.

color values to quickly stabilize and hence, color values do not change much. However, in *RCP*, as color values are received through refresh, the stabilized colors do get changed, in turn leading to more changes and hence, stale values. Using stale values slows down the convergence of these algorithms which can be seen from the data in Figure 2.9. On an average, *RCP* requires 49.5% more iterations than *SCP+RW* while *Stale-2* and *Stale-3* require 146.4% and 176.2% more iterations than *SCP+RW*. Note that *Stale-n* versions for NP did not terminate within 5 times the time required when run with *SCP+RW*; hence, for these cases, we do not show the data for remote fetches, iterations, and staleness values.

It is interesting to note that even though *Stale-n* effectively tries to avoid remote fetches on the critical path, it is often done at the cost of delaying convergence. This delay in convergence, in turn, results in more remote fetches. Thus, the overall performance can be adversely affected (e.g., HS, PR). On the other hand, even though *Stale-n* sometimes converges in nearly same number of iterations (e.g., WS), the overall performance gains are less when compared to benefits achieved from *RCP*. This is mainly because the computation

thread often blocks when the staleness of local objects crosses beyond the threshold n and hence the reduction in remote fetches is not as significant as in *RCP*. This interplay between reduction in remote fetches and increase in the number of iterations for convergence makes the best choice of n to be specific for each benchmark. This can be clearly observed by comparing WS and PR: *Stale-2* performs better than *Stale-1* for WS whereas the latter performs better than the former for PR. Hence, in *Stale-n*, selecting the value of n is benchmark specific and hard to do. *RCP* releases users from such concerns and outperforms *Stale-n* in each case.

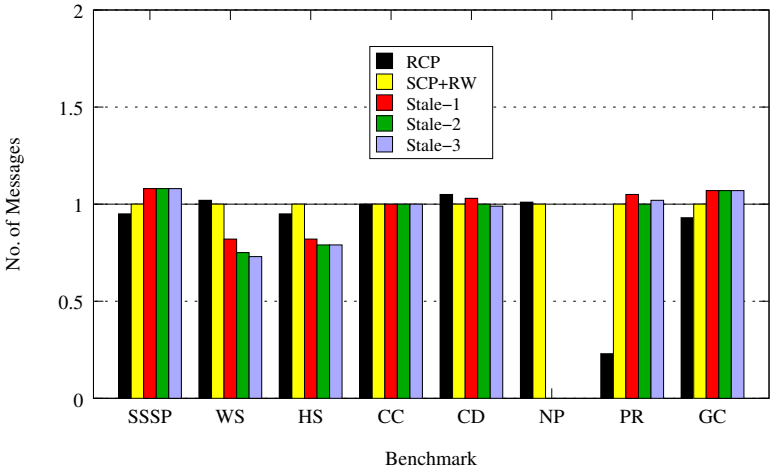


Figure 2.10: Number of protocol messages normalized wrt SCP+RW.

Finally, we compare the communication overhead of *RCP*, *Stale-n*, and *SCP+RW* by measuring the number of protocol messages required in each case. As shown in Figure 2.10, in most cases, *RCP* requires fewer protocol messages compared to *SCP+RW* and in the remaining cases, it requires less than 5% additional messages. This is mainly because there is a drastic reduction in the number of remote fetch requests for *RCP* (as seen in Figure 2.7), most of which are reduced to asynchronous refresh requests. PR using *RCP*

requires only 22.8% messages of that required by *SCP+RW* because invalidate and refresh messages are far fewer than the reduction in the remote fetch requests. Even though WS and HS also experience a similarly large reduction in remote fetch requests using *RCP*, they require sufficiently more invalidate and refresh messages which leads to their overall communication costs to be nearly same as *SCP+RW*.

2.5.3 Design Choices of RCP

To better understand our design choices for the relaxed consistency protocol, we evaluate the protocols using synthetic benchmarks that exploit different application specific properties. The synthetic benchmarks are designed similar to other benchmarks which mainly fetch neighboring vertices and compute new vertex values. The data is based upon the *HiggsTwitter* graph input and the programs were run for a pre-configured number of iterations.

Piggy-backing Updates vs. RCP Invalidates. Figure 2.11 shows the effect of allowing multiple writes on *RCP*, *SCP* and *SCP* with piggy-backed updates (**SCP+PB**) where the updates are sent to remote nodes along with invalidation messages. The execution times are normalized with respect to configurations where an object is written once per iteration. We observe that even though *SCP+PB* performs better than *SCP*, the execution times for both the configurations increase drastically when objects are written more often in an iteration. It is interesting to note that the benefits achieved from *SCP+PB* over *SCP* reduce as the number of times objects are written per iteration increases; this happens because redundant updates are sent by *SCP+PB* which becomes costly. On the other hand,

RCP easily tolerates multiple writes and consistently performs similar to the baseline. The resulting execution times normalized with respect to *SCP* are shown in Figure 2.12.

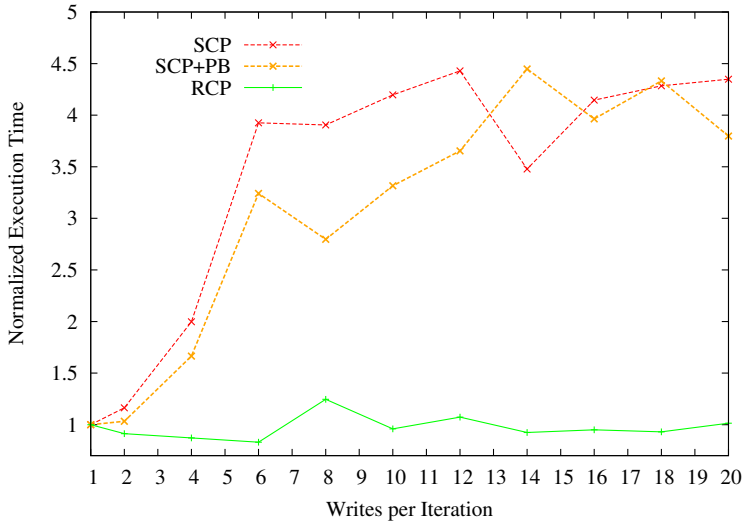


Figure 2.11: Execution times of SCP with and without piggy-backed updates and RCP for different write configurations normalized wrt single write versions.

Sensitivity of RCP to Object Sizes. In Figure 2.13, we compare the performance of *SCP*, *SCP+PB*, and *RCP* for different object sizes. Object sizes are varied by adding a bloat to the base object which, by itself, only consists of a double value (8 bytes). We can see that sending updates with invalidates performs similar to when only invalidates are sent and *RCP* consistently performs better than the other 2 configurations.

Sensitivity of RCP to Communication Delay. In Figure 2.14 we show the impact of fetch latency on the maximum staleness of object values used for computations. We observe that as the fetch latencies are increased, the maximum staleness varies. As expected, we notice that most of the objects have low staleness values, leaving very few objects which

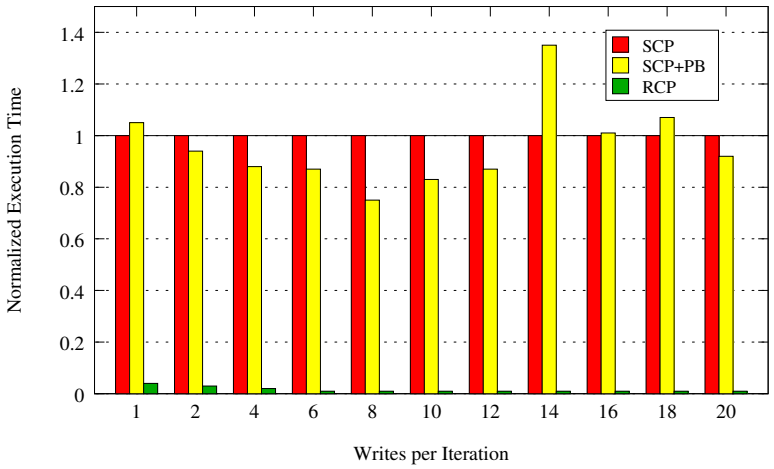


Figure 2.12: Execution times of SCP with and without piggy-backed updates and RCP for different write configurations normalized wrt SCP.

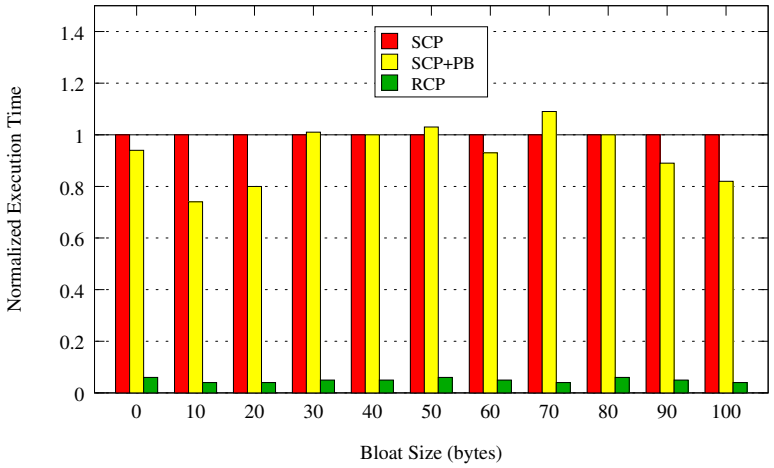


Figure 2.13: Execution times of SCP with and without piggy-backed updates and RCP for different object sizes.

are very stale. Hence, it is important to control the staleness using an upper bound which avoids potential staleness runaways in such cases.

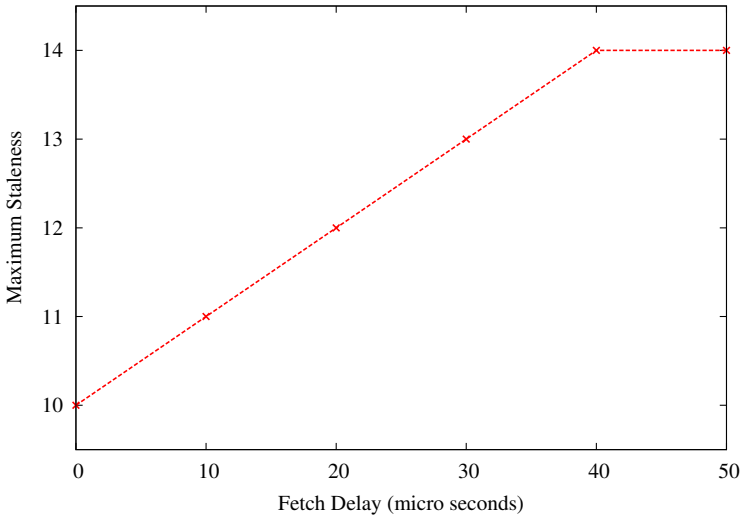


Figure 2.14: Maximum staleness for objects used in RCP with varying communication delay.

		SSSP	PR	GC	CC	NP
Orkut	**RCP**	161	822	92	90	2.35
	GraphLab	239	829	248	102	140
LiveJournal	**RCP**	21	343	17	22	133
	GraphLab	15	295	X	66	150
Pokec	**RCP**	9.47	169	8.81	7.1	1.74
	GraphLab	8.7	159	173	40	76
HiggsTwitter	**RCP**	2.5	15	3.59	4.1	0.48
	GraphLab	5.5	X	263	16	32
RoadNetCA	**RCP**	49	7.70	0.93	56	16
	GraphLab	60	88	50	220	37
RoadNetTX	**RCP**	44	5.05	0.53	50	15
	GraphLab	18	78	60	115	X

Table 2.5: Execution times (in sec) of SSSP, PR, GC, CC, and NP using RCP and GraphLab (GL) on a 16-node cluster. An x indicates that execution did not complete either because it crashed or continued for over 60 minutes.

2.5.4 Comparison with Other Systems

To exhibit the effectiveness of exploiting asynchrony, we compare the performance of asynchronous algorithms running using *RCP* with the performance of the popular *bulk*

synchronous parallel (BSP) model as it is supported by existing graph processing frameworks. In addition, we also compare the performance of our system with GraphLab [85], a popular distributed graph processing framework.

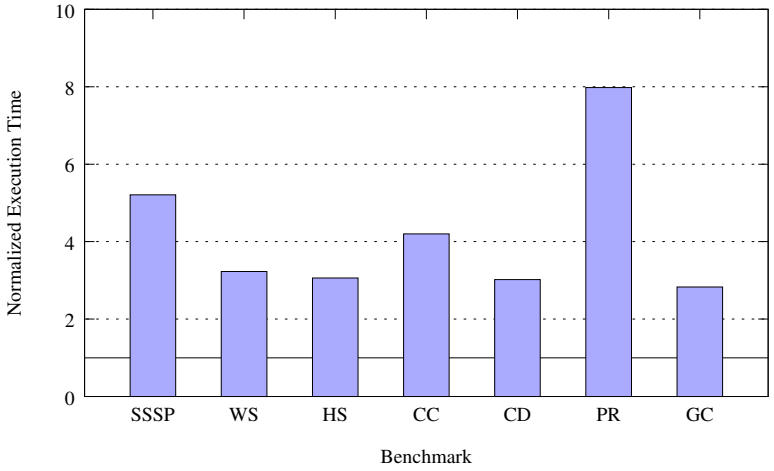


Figure 2.15: Execution times for BSP based implementations normalized wrt their asynchronous versions that use RCP.

RCP vs. Bulk Synchronous Parallel. The Bulk Synchronous Parallel (BSP) is a computation model that was proposed to efficiently parallelize applications on a set of processors [126]. Algorithms based on this model perform a series of supersteps where each superstep is composed of the following phases: *Computation* - multiple processes and threads concurrently execute several computations using locally available data values; *Communication* - the computed data values are made available to required processes and threads; and *Synchronization* - a barrier is executed by all the processes to conclude the superstep.

Recent advances in developing generic frameworks for parallel applications heavily rely on this model. However, by maintaining a separate communication phase, the updated

values are forcibly propagated throughout the system, which unnecessarily introduces significant overhead for asynchronous algorithms.

Figure 2.15 shows the execution times of *BSP* based implementations for our benchmarks normalized with respect to their corresponding asynchronous versions that make use of our proposed protocol. On an average, our asynchronous algorithms are faster than *BSP* based algorithms by 4.2x. Apart from PR, *BSP* versions take 2.8x to 5.2x more than their asynchronous versions using our protocol. PR takes 7.9x more time with *BSP* mainly because it spends more time in the communication phase compared to other benchmarks. Again, *BSP* version for NP did not terminate within 10 times the time required when run with *RCP* and hence, we do not show its performance.

Comparison with GraphLab. GraphLab [85] is a popular graph processing framework which is closest to our work because it provides shared memory abstractions to program over a distributed environment. We compare the performance of GraphLab with *RCP* using five benchmarks – SSSP, PR, GC, CC, and NP – four of which are provided in the graph analytics toolkit distributed with the GraphLab software. Since GraphLab provides both synchronous and asynchronous versions of some of these programs, we report the best times obtained here. In order to have a fair comparison, similar to replication of boundary vertices in GraphLab, caches were pre-populated with replicas of boundary vertices to eliminate latencies incurred by cache warmups.

In Table 2.5 we report the absolute execution times (in sec) for SSSP, PR, GC, CC, and NP using RCP and GraphLab for the four power law graph inputs, as GraphLab has been designed to efficiently handle such graphs. The relative performance of GraphLab

and *RCP* varies significantly across inputs and benchmarks. We observe that for the Orkut and HiggsTwitter inputs *RCP* consistently outperforms GraphLab. For the LiveJournal and Pokec inputs, GraphLab provides superior performance for SSSP and PR. Finally, for GC and CC benchmarks *RCP* consistently outperforms GraphLab across different inputs. Overall, the performance of *RCP* compares favorably with GraphLab for power law graphs. It should be noted that *RCP* is based on the the Relaxed Consistency Model which is orthogonal to GraphLab's consistency models. Hence, this model can also be incorporated in GraphLab.

Although GraphLab has been designed primarily for power law graphs, we did test it for other inputs. As shown in Table 2.5, on the RoadNetTX graph the above benchmarks took 0.5 sec to 50.9 sec using *RCP* and 18.3 sec to 115.6 sec on GraphLab. We also coded other benchmarks on GraphLab and compared their performance for different inputs. For both NP and WS, *RCP* consistently outperformed GraphLab.

2.6 Summary

In this chapter, we demonstrated an effective solution for exploiting the asynchronous nature of iterative algorithms for tolerating communication latency in a DSM based cluster. We designed a relaxed object consistency model and the RCP protocol. This protocol tracks staleness of objects, allows threads to utilize stale values up to a given threshold, and incorporates a policy for refreshing stale values. Together, these features allow an asynchronous algorithm to tolerate communication latency without adversely impacting algorithm's convergence. We demonstrated that for a wide range of asynchronous

graph algorithms, on an average, our approach outperforms: prior relaxed memory models that allow stale values by at least 2.27×; and BSP model by 4.2×. In the next chapter, we will see how asynchrony can be leveraged to efficiently handle machine failures that can occur during processing.

Chapter 3

Confined Recovery

In the previous chapter, we saw how asynchrony can be exploited by relaxing consistency to expose different legal executions based on different combination of stale values. While such a processing model improved the overall performance, distributed systems are susceptible to machine failures and hence, it is important for the graph processing systems to incorporate fault tolerance mechanisms to handle such failures. In this chapter, we develop an efficient fault tolerance technique based on the key observation that the asynchronous model allows multiple legal executions and hence, it is acceptable to recover the execution state such that it is legal under the asynchronous model.

Fault tolerance in distributed graph processing systems is provided by periodically snapshotting the vertex/edge values of the data-graph during processing, and restarting the execution from the previously saved snapshot during recovery [86, 135, 96]. The cost of fault tolerance includes: overhead of periodic checkpoints that capture *globally consistent snapshots* [19] of a graph computation; and *repeating computation* whose results are

discarded due to the roll back during the recovery process. For synchronous graph processing systems, solutions that lower these overheads have been proposed [86]. However, development of efficient fault tolerance techniques for asynchronous graph processing lags behind. To perform recovery, asynchronous processing systems roll back the states of *all the machines* to the last available snapshot and resume the computation from that point. This is because of inherent non-determinism in asynchronous processing which discards the possibility of reconstructing lost execution state using the saved inputs. Hence, fault tolerance in asynchronous graph processing systems has the following two drawbacks:

- *Redundant computation*: Since recovery rolls back the states of *all* machines to the latest snapshot, when processing is resumed, the computation from snapshot to the current state is repeated for machines that did not fail.
- *Increased network bandwidth usage*: The local snapshots are saved on remote machines, either on the distributed file system or in memory. All machines bulk transfer snapshots over the network simultaneously. This stresses the network and increases peak bandwidth usage. The problem gets worse in case of a multi-tenant cluster.

In this chapter, we leverage the availability of multiple legal executions due to inherent asynchrony in order to quickly construct a legal execution state (upon failure) that may be different from the prior execution states. We construct such an alternate state by rolling back the states of only the failed machines, hence not impacting the progress made by other machines that did not fail.

We present CoRAL, a highly optimized recovery technique for asynchronous graph processing that is the first *confined recovery* technique for asynchronous processing. We

observe that the correctness of graph computations using asynchronous processing rests on enforcing the *Progressive Reads Semantics* (PR-Semantics) which results in the graph computation being in PR-Consistent State. Therefore recovery need not roll back graph state to a globally consistent state; it is sufficient to restore state to a PR-Consistent state. We leverage this observation in two significant ways. First, non-failing machines do not rollback their graph state, instead they carry out *confined recovery* by reconstructing the graph states of failed machines such that the computation is brought into a PR-Consistent state. Second, globally consistent snapshots are no longer required, instead *locally consistent snapshots* are used to enable recovery.

Finally, we demonstrate using real-world graphs show that our technique recovers from failures and finishes processing 1.5× to 3.2× faster compared to the traditional asynchronous checkpointing and recovery mechanism when failures impact 1 to 6 machines of a 16 machine cluster. Moreover, capturing locally consistent snapshots significantly reduces intermittent high bandwidth usage required to save the snapshots – the average reduction in 99th percentile peak bandwidth ranges from 22% to 51% while 1 to 6 snapshot replicas are being maintained.

The remainder of the chapter is organized as follows. Section 3.1 discusses the traditional globally consistent snapshot model and the recovery strategies used by various systems. Section 3.2 introduces the confined recovery and locally consistent snapshot model. We discuss the implementation of our prototype, experimental setup, and results of evaluation in Section 3.3.

3.1 Background and Motivation

Distributed graph processing systems provide fault tolerance to handle machine failures that can occur in the midst of a graph computation. The failure model assumes *fail-stop* failures, i.e. when a machine fails, it does not lead to malicious/unexpected behavior at other machines. Once a machine in a cluster fails, its workload can be distributed across remaining machines in the cluster or the failed machine can be replaced by another server.

Fault tolerance is provided via *checkpointing* and rollback based *recovery* mechanism [36]. The checkpoints, that are performed periodically, save a *globally consistent snapshot* [19] of the state of a graph computation. A captured snapshot represents the state of the entire distributed graph computation such that it includes a valid set of values from which the processing can be resumed. Therefore *recovery*, that is performed when a machine fails, rolls back the computation to the latest checkpoint using the saved snapshot and resumes execution. For capturing a globally consistent snapshot, both synchronous and asynchronous checkpointing methods exist [87]. Synchronous checkpointing suspends all computations and flushes all the communication channels before constructing the snapshot by capturing the graph computation state at each machine whereas asynchronous checkpointing incrementally constructs the snapshot as the computation proceeds. The frequency of capturing snapshots balances checkpointing and recovery costs [139].

A *globally consistent snapshot* of a distributed graph computation is defined below.

Definition 3.1.1. Given a cluster $C = \{c_0, c_1, ..., c_{k-1}\}$, a *Globally Consistent Snapshot* of C is a set of local snapshots, denoted as $S = \{s_0, s_1, ..., s_{k-1}\}$, such that it satisfies [P-LCO] and [P-GCO], as specified below.

[P-LCO]: Each $s_i \in S$ represents a consistent state of the subgraph processed by $c_i \in C$ (i.e., vertex/edge values iteratively computed by c_i) that is computable by the processing model from the initial state of the graph.

[P-GCO]: S represents a globally consistent state of the entire graph that is computable by the processing model from the initial state of the graph.

Note that a local snapshot of a machine simply consists of vertex/edge values computed by that machine; the structure of the graph, along with any static vertex and edge values, are not captured in the snapshot because they remain the same throughout the computation.

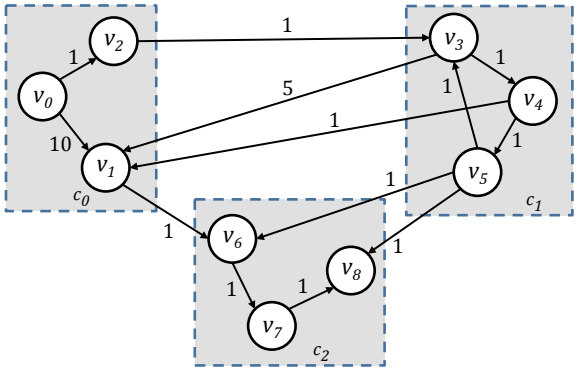


Figure 3.1: Example graph.

	c_0			c_1			c_2		
Step	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
t_0	0	∞	∞	∞	∞	∞	∞	∞	∞
t_1	0	10	1	∞	∞	∞	∞	∞	∞
t_2	0	10	1	2	∞	∞	11	∞	∞
t_3	0	7	1	2	3	∞	11	12	∞
t_4	0	4	1	2	3	4	8	12	13
t_5	0	4	1	2	3	4	5	9	5
t_6	0	4	1	2	3	4	5	6	5

Table 3.1: SSSP example.

	Execution Model		Checkpointing Model		Checkpointing Consistency Model		Recovery Model	
	Sync	Async	Sync	Async	Globally Consistent	Locally Consistent	All Rollback	Minimal Rollback
Pregel	✓	✗	✓	✗	✓	✗	✗	✓
GraphLab	✓	✓	✓	✓	✓	✗	✓	✗
GPS	✓	✗	✓	✗	✓	✗	✓	✗
GraphX	✓	✗	✓	✗	✓	✗	✓	✗
Imitator	✓	✗	Replication		Consistent Replication		None	
Zorro	✓	✗	None		None		✗	✓
CoRAL	✗	✓	✗	✓	✗	✓	✗	✓

Table 3.2: Key characteristics of existing graph processing systems and our CoRAL system.

Consider the graph shown in Figure 3.1. Vertices v_0 through v_8 are partitioned across machines c_0, c_1, and c_2. After partitioning, the edges that cross machines translate into *remote* reads. A vertex having at least one neighbor residing on a different machine is called a *boundary* vertex – in this example, vertices v_1 and v_2 are boundary vertices of machine c_0. The boundary vertices are usually replicated on remote machines so that they are readily available to remote neighbors for their computation. Table 3.1 shows how computation of shortest paths proceeds with v_0 as source. The table shows steps t_0 through t_6 of the computation. Let us assume that a globally consistent checkpoint captures (highlighted) values at step t_3. Now, if c_2 fails at t_5, instead of starting from the state at t_0 (first row), all the execution states are rolled back to the snapshot taken at t_3 and the processing is resumed from this rolled back state.

Next we summarize the pros and cons of existing checkpointing and recovery methods to motivate our approach.

(1) Synchronous processing systems like Pregel, GPS [105], GraphLab's synchronous model, and Trinity's synchronous model [109] use synchronous checkpointing that captures globally consistent snapshot by initiating the checkpointing process at the start of a global iteration (super step). Therefore the values captured are values that exist at the beginning of a global super step. Pregel performs ***confined recovery*** that requires rolling back the state of only the failed machine as follows. After capturing a snapshot, at each machine, the inputs read from other machines for boundary vertices are saved so that they can be replayed during recovery to construct the execution state of the failed machine at the point of failure. Trinity models confined recovery using buffered logging while [111] performs confined recovery in a distributed manner.

Two additional approaches for *fast recovery* have been proposed. Zorro [96] is motivated by the observation in other works (GraphX [45], Giraph [25], and Distributed GraphLab [84]) that users often disable the fault tolerance to accelerate processing. Thus it chooses to discard the checkpointing process altogether to eliminate its overheads. Upon failures, the recovery process constructs an approximate execution state using the replicated boundary vertices residing on remaining machines. Hence, it achieves fast recovery at the cost of sacrificing accuracy [96]. Imitator [135] maintains in-memory replicated globally consistent execution state throughout the execution so that recovery from failure is immediate. The cost of this approach is the overhead of maintaining consistent replicas in memory all the time. Finally, GraphX [45] relies on Spark [142] for tracking the lineage of data in memory, i.e., saving the intermediate results of high-level operations over data; however, when the lineage tree becomes very large, it resorts to checkpointing.

Although for synchronous processing systems recovery has been optimized, their overall performance can be significantly lower than that of asynchronous processing systems [27, 129]. Next, we discuss asynchronous systems.

(2) Asynchronous processing systems like GraphLab's asynchronous model uses asynchronous checkpointing technique that captures the vertex/edge values by developing a mechanism based on the Chandy-Lamport snapshot algorithm [19] that is also used in other domains like Piccolo [95]. While the snapshots captured by such asynchronous checkpointing reflect values coming from global states at different centralized clock times, the order in which the values are captured with respect to communication and computation performed guarantee that the snapshot is globally consistent. Trinity's asynchronous model, on the other hand, interrupts execution to capture the global snapshot.

While Pregel's confined recovery is useful as it only rolls back the state of the failed machine, it is applicable only for synchronous processing environments since the order in which iterations progress is deterministic. For asynchronous execution, the lost execution state cannot be reconstructed using the saved inputs because the order in which vertex computations observe and process the input values varies over the execution, thus making this technique inapplicable. As a result, to perform recovery, asynchronous processing systems roll back the states of *all the machines* to the last available snapshot and resume the computation from that point. This approach has following two drawbacks:

– *Redundant computation*: Since recovery rolls back the states of *all* machines to the latest snapshot, when processing is resumed, the computation from snapshot to the current state is repeated for machines that did not fail.

– *Increased network bandwidth usage*: The local snapshots are saved on remote machines, either on the distributed file system or in memory. All machines bulk transfer snapshots over the network simultaneously. This stresses the network and increases peak bandwidth usage. The problem gets worse in case of a multi-tenant cluster.

Summary. Table 3.2 summarizes the key characteristics of above frameworks. The existing systems rely upon globally consistent snapshots for recovery. Apart from the synchronous solutions of Pregel and Zorro, none of the works perform minimal rollback. GraphLab's asynchronous engine captures globally consistent snapshots and rolls back the state of all the machines. Solutions that do not rely on recovery via rollback to a checkpoint, either incorporate consistent replication (Imitator) or relax the correctness guarantees that leads to imprecise results in case of failures (Zorro).

3.2 Confined Recovery for Asynchronous model via Lightweight checkpointing

The goal of our work is to develop a technique that: (1) uses *asynchronous* processing model as it provides high performance; (2) performs *minimal* rollback and avoids network bandwidth problem due to checkpointing; and (3) achieves complete recovery so that the final solutions are guaranteed to be *accurate*. Next we present **CoRAL**, a **Co**nfined **R**ecovery technique for iterative distributed graph algorithms being executed under the **A**synchronous processing model that uses **L**ightweight checkpoints.

Characteristics of Asynchronous Graph Computation. Under the asynchronous model [129, 131], graph computations are inherently non-deterministic because the model relaxes read-write dependences to allow machines to use stale values of remote vertices such that all machines can continue processing independently by skipping intermediate updates from remote vertices. As a consequence, under the asynchronous model, there are multiple legal executions, all of which upon convergence produce the same final results.

Asynchronous execution typically orders the values read for each vertex x via the *Progressive Reads Semantics* (PR-Semantics) such that over time, x is assigned different values $v(x,0), v(x,1), \cdots, v(x,n)$ by the machine on which it resides and these values are used (read) during processing on other machines.

Definition 3.2.1. *PR-Semantics* ensures that if a read of x performed by a thread observes the value $v(x,i)$, the subsequent read of x by that same thread must observe value $v(x,j)$ such that it either satisfies [V-SAM] or [V-FUT] as given below:

[V-SAM]: $j = i$, that is, the same value is observed; or

[V-FUT]: $j > i$, that is, a fresher value is observed on the second read.

This means that, once a value for any data item is read by a thread, no earlier values of that data item can be read by the same thread. The PR-Semantics ensures that each thread observes the values for a given data item in the same order as they were produced, and hence, convergence and correctness of asynchronous algorithms can be reasoned about.

Definition 3.2.2. An execution state $E = \{e_1, e_2, ..., e_{k-1}\}$ of a graph computation is *PR-Consistent* if it is reached by performing the graph computation using an asynchronous processing model that follows the PR-Semantics.

Thus, following a failure, the recovery process must construct the state of the subgraph(s) lost due to machine failure(s) such that the resulting computation is in a PR-Consistent state. Resuming execution from a PR-Consistent state guarantees that all future remote reads adhere to PR-Semantics.

The reliance of graph processing algorithms on *PR-Semantics* and *PR-Consistent* state can be found in literature. In [37] the self-stabilizing nature of PageRank algorithm is proven assuming that the underlying system guarantees progressive reads. Below we derive an equivalence between a PR-Consistent execution and a legal asynchronous bounded staleness based execution [129, 131].

Theorem 3.2.1. *Every PR-Consistent execution state of graph computation starting from an initial state I is equivalent to an execution state under some legal staleness based asynchronous execution* [129, 131] *starting from I.*

Proof. The full PR-Consistent execution can be viewed as a sequence of intermediate PR-Consistent execution states $E_0 \rightarrow E_1 \rightarrow E_2 \rightarrow ... \rightarrow E_n$ starting at the initial state $I = E_0$. Hence, we prove this theorem using induction on $E_i (0 \leq i \leq n)$.

Base Case ($E_i = E_0 = I$): E_0 is PR-Consistent since no reads are performed. It is the same starting execution state for staleness based asynchronous execution.

Induction Hypothesis ($E_i = E_k, k > 0$): E_k is PR-Consistent and is equivalent to an execution state under some legal staleness based asynchronous execution.

Induction Step ($E_i = E_{k+1}$): E_{k+1} is a PR-Consistent execution state constructed after E_k based on values read from vertices in V_{k+1}. Without loss of generality, let us consider a vertex $x \in V_{k+1}$ whose latest value read prior to computation of E_{k+1} is $v(x, p)$. When

performing a computation requiring the value of x, the *current value* of x is first read and then used in the computation. From Definition 3.2.1, we know that the value read for computation of E_{k+1} is $v(x,q)$ such that $q \geq p$. Between the reading of this value and its use, the value of x can be changed to $v(x,r)$ by another computation, i.e., $r \geq q$. This leads to the following cases:

– Case 1 ($q = r$): The second read returned $v(x,q) = v(x,r)$. In the equivalent staleness based execution, this read is considered to have returned the *current* or the *freshest* value available for x, and hence is usable, which results in the same computation being performed.

– Case 2 ($q < r$): The second read returned $v(x,q) \neq v(x,r)$. In the equivalent staleness based execution, such a value is considered to be *stale* by $r-q$ versions, and is still usable in the asynchronous model where staleness bound b^x_{k+1} is at least $r-q$. Note that the staleness bound does not impact correctness, it merely impacts the performance of asynchronous execution [129].

The above reason can be applied to all the vertices in V_{k+1} whose values are used to compute E_{k+1}. Let $s = \max_{0>j\leq k+1}(\max_{x\in V_j}(\ b^x_j\ )\ )$ be the maximum staleness of reads across all the vertex values read in Case 2. Across all the possibilities, the computations in PR-Consistent execution and an asynchronous staleness based execution with staleness bound of at least s (i.e., $\geq s$) are equivalent since they are based on same values, and hence, they result in the equivalent or same execution state E_{k+1}. □

Corollary 3.2.1. *The final execution state reached by a PR-Consistent execution is equivalent to the final execution state under some legal staleness based asynchronous execution* [129, 131].

To further illustrate the efficacy of *PR-Semantics*, we consider SSSP, a popular example of monotonic graph algorithms (other examples include Connected Components, K-Core, etc.) where vertex values exhibit *monotonicity* which cannot be preserved without *PR-Semantics*. Table 3.3 shows the effect of violating the PR-Semantics at t_5 in our SSSP example from Figure 3.1. If at t_5, c_0 observes the old value of $v_4 = \infty$ after having observed $v_4 = 3$ at t_4, the value of v_1 is computed as 7 via v_3 as shown below.

$$path(v_1) = min(path(v_3) + weight(v_3, v_1),$$
$$path(v_4) + weight(v_4, v_2))$$
$$= min(2 + 5, \infty + 1) = 7$$

	c_0			c_1			c_2		
Step	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
t_0	0	∞	∞	∞	∞	∞	∞	∞	∞
t_1	0	10	1	∞	∞	∞	∞	∞	∞
t_2	0	10	1	2	∞	∞	11	∞	∞
t_3	0	7	1	2	3	∞	11	12	∞
t_4	0	4	1	2	3	4	8	12	13
t_5	0	7	1	2	3	4	5	9	5

Table 3.3: Violation of PR-Semantics disrupting monotonicity in SSSP.

This violates the monotonicity property of SSSP because the shortest path value for v_2, instead of decreasing, increases from 4 to 7.

Overview of PR-Consistent Recovery. The above characteristics of asynchronous graph processing lead to new more *relaxed notion of recovery*, called *PR-Consistent* recovery, that allows use of confined recovery using lightweight checkpoints. Its key features follow.

(1) *Confined Recovery*: Given s_f and c_f such that s_f is the state of the subgraph on machine c_f just before it fails. The state of the subgraph on c_f following recovery, say s_r, need not be the same as s_f. However, both s_f and s_r must correspond to legal executions during which the PR-Semantics is preserved. We exploit this flexibility to achieve *Confined Recovery*, i.e. the subgraph states at non-failing machines is not rolled back.

(2) *Lightweight Checkpoints*: Deriving the recovered state s_r does not require globally consistent snapshots. It simply requires periodically taken local snapshots of all machines which we refer to as *Locally Consistent Snapshots*. The global ordering across the local snapshots, called *PR-Ordering*, must be captured to enforce PR-Semantics during confined recovery for *multiple machine failures*. The sufficiency of locally consistent snapshots solves the problem of *increased network bandwidth usage* due to bulk network transfer for saving snapshots during checkpointing. The decision to capture a local snapshot at a given point in time can be made either by a central coordinator to minimize the number of snapshots being simultaneously saved, or locally by the machine in which snapshot is to be captured.

(3) *Fast Recovery*: Once a machine in a cluster fails, its workload is distributed across remaining machines in the cluster which then collectively reconstruct the state s_r in parallel. To further reduce checkpointing overhead and speedup recovery, the replicated snapshots are stored in-memory on remote machines. Both of these design decisions are based on RAMCloud's approach [93] for fast replication and recovery; however, our technique is applicable if a failed machine is replaced by a cold server and snapshots are stored on a distributed file system.

In summary, CoRAL captures light-weight Locally Consistent Snapshots and PR-Ordering information that allow the recovery of the state(s) corresponding to failed machine(s) such that reconstructed state is PR-Consistent from which execution can be correctly resumed.

3.2.1 PR-Consistent Recovery: *Single Failure Case*

For ease of understanding, in this section we show how PR-Consistent state is restored in case of a single machine failure and in the next section we present the additions required to handle multiple simultaneous machine failures.

We introduce the concept of *Locally Consistent Snapshots* and then present a recovery algorithm that uses them to construct the *PR-Consistent* state following a failure. Since a globally consistent checkpoint captures a valid graph state, following a failure, rolling back entire graph state to such a captured state is sufficient to restore execution to a PR-Consistent state. However, restoring state via a globally consistent snapshot is too strong of a requirement, i.e. it is not necessary for satisfying PR-Semantics after recovery. In fact, allowing global inconsistencies in the captured graph state is acceptable due to the relaxed nature of asynchronous execution model semantics.

A *locally consistent checkpoint* represents this relaxed notion of a distributed snapshot. Next we define a locally consistent snapshot of the system.

Definition 3.2.3. Given a cluster $C = \{c_0, c_1, ..., c_{k-1}\}$, a *Locally Consistent Snapshot* $S = \{s_0, s_1, ..., s_{k-1}\}$, is defined as a set of local snapshots such that it satisfies [P-LCO] as specified below.

[P-LCO]: Each $s_i \in S$ represents a consistent state of the subgraph processed by $c_i \in C$ that is computable by the processing model from the initial state of the graph.

Note that locally consistent checkpoints do not enforce consistency requirement across different local snapshots and hence, eliminate the need to save snapshots at the same time; by staggering their collection over time, the stress on network bandwidth is lowered. Also, since failures can occur while a snapshot is being captured and transferred to remote machines, the snapshot should not be committed until the entire snapshot has been received.

The recovery process has two primary goals: first, the execution state should be restored to a PR-Consistent state; and second, the execution state of the machines which are not affected by failures must not be rolled back, i.e., the recovery process should be confined to workload of the failed machine.

Formally, let $E^c = \{e_0^c, e_1^c, ..., e_i^c, ..., e_{k-1}^c\}$ represent the latest execution state of machines in C right before failure of a single machine $c_i \in C$. Due to failure, the local execution state e_i^c is lost and the remaining available execution state is $E^{cf} = E^c \setminus \{e_i^c\}$. The recovery process must reconstruct the local execution state e_i^r of c_i such that, $E^r = E^{cf} \cup \{e_i^r\}$ represents a PR-Consistent state while, e_i^r may be different from e_i^c. Figure 3.2 shows this recovery process – when e_i^c is lost, the subgraph is processed using values from s_i and available inputs from E^c (i.e., E^{cf}) to generate e_i^r.

Next we consider the recovery algorithm. Let s_i be the last snapshot captured for e_i during checkpointing. Naïvely constructing e_i^r by directly using values from s_i does not represent a PR-Consistent state because $\forall e_j^c \in E^{cf}$, the values in e_j^c can be based on fresher values from c_i which became available after capturing s_i and hence, further reads from

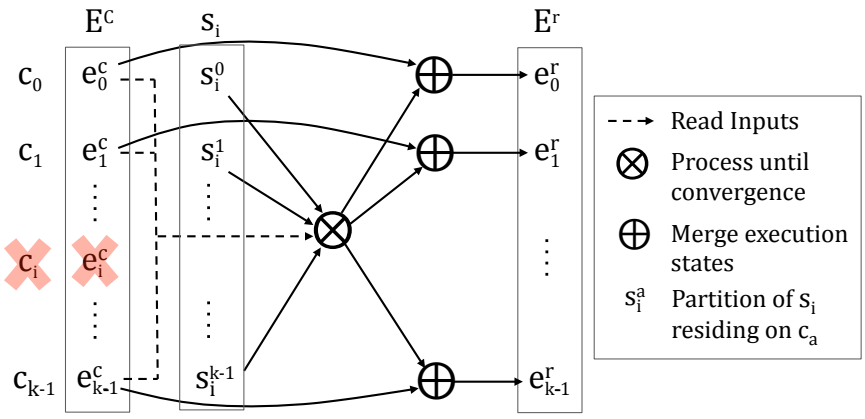


Figure 3.2: Recovery from single failure.

e_i^r will violate the PR-Semantics. We use $=_{PR}$ to denote the PR-Consistent relationship between two local execution states, i.e., if e_a and e_b are PR-Consistent, then $e_a =_{PR} e_b$. Hence, we want to construct e_i^r such that $\forall e_j^c \in E^{cf}$, $e_i^r =_{PR} e_j^c$.

Algorithm 2 Recovery from single failure.

```
1: s_i: Snapshot of failed machine c_i
2: E^cf: Current execution state of remaining machines
3: function RECOVER ( )
4:     e_i ← LOADSUBGRAPH(s_i)
5:     READBOUNDARYVERTICES(e_i, E^cf)
6:     e_i^r ← PROCESSUNTILCONVERGENCE(e_i)
7:     E^r ← E^cf ∪ {e_i^r}
8:     return E^r
9: end function
```

Algorithm 2 constructs a PR-Consistent state E^r. The algorithm first loads the subgraph which was handled by the failed machine and initializes it with: values from s_i (line 4); and current values of boundary vertices coming from E^{cf} (line 5). Note that this initialization of boundary vertex replicas does not violate PR-Semantics because values in s_i are based on older values of boundary vertices which were available when s_i was captured. Then the created subgraph e_i is iteratively processed in isolation until convergence (line

6) – this is the crucial step in the algorithm. Fully processing e_i ensures that the effects of fresher boundary vertex' values are fully propagated throughout the subgraph. Hence, the values in e_i^c before failure were either older than or at most same as the values in e_i^r. This means, any further reads from e_i^r performed by any $e_j^c \in E^{cf}$ return fresher values and hence, do not violate PR-Semantics, i.e., $\forall e_j^c \in E^{cf}$, $e_i^r =_{PR} e_j^c$. Hence, e_i^r is included in E^{cf} (line 7) to represent the PR-Consistent state E^r which is used to resume processing.

3.2.2 PR-Consistent Recovery: *Multiple Failures*

Recovering from a failure impacting multiple machines introduces an additional challenge. To recover a PR-Consistent state, we must ensure that the recovery process operates on the snapshots of failed machines such that it does not violate the PR-Semantics. This means, the PR-Consistent state must be constructed by carefully orchestrating the *order* in which snapshots are included and processed for recovery. Hence, we introduce the concept of *PR-Ordering of Local Snapshots*, which is required to carry out PR-Consistent confined recovery following failure of multiple machines.

PR-Ordering of Local Snapshots. To recover a state after which any further reads will adhere to progressive reads semantics, we must capture the read-write dependences between data elements across different local snapshots that were truly imposed due to *PR-Semantics*. Capturing this information at the level of each data item is expensive due to two reasons: 1) the space of the snapshots blows up with number of inter-machine dependencies, which in graph processing is based on the edge-cut; and 2) capturing such information requires synchronization between the local machine and all other machines.

PR-Semantics naturally enforces dependency ordering across data values. We lift this dependency ordering to a higher level of abstraction – the local snapshots. If we only track the ordering incurred due to progressive reads across the local snapshots, the amount of information maintained per snapshot reduces drastically. This brings us to the definition of ordering of local snapshots.

A *PR-Ordering* of snapshots, denoted as $\leq_{PR}$, defines an ordering across local snapshots based on the order of reads performed by data-elements within the snapshot.

Definition 3.2.4. The *PR-Ordering* of a pair of local snapshots, denoted as $s_i \leq_{PR} s_j$, indicates that the values in s_i for machine c_i were computed based on values from machine c_j which were available no later than when s_j for c_j was captured.

In other words, $s_i \leq_{PR} s_j$ ensures that values captured in s_i were based on reads of data from c_j prior to capturing s_j. This naturally leads us to the following observation. While PR-Ordering is a pairwise ordering across local snapshots, we prove that a total PR-Ordering is required to perform recovery of PR-Consistent state.

Formally, let $E^c = \{e_0^c, e_1^c, ..., e_{k-1}^c\}$ represent the latest execution state of machines in C right before failure of machines in $F \subset C$. Let $E^l = \{e_i^c \mid c_i \in F\}$ be set of local execution states lost due to failure, leaving the remaining available execution state to be $E^{cf} = E^c \setminus E^l$. The goal of the recovery is to reconstruct the set of local execution states $E^{rf} = \{e_i^r \mid c_i \in F\}$ of failed machines such that, $E^r = E^{cf} \cup E^{rf}$ represents a PR-Consistent state while, $\forall c_i \in F$, e_i^r may be different from e_i^c.

	c_0			c_1			c_2		
Step	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
t_0	0	∞	∞	∞	∞	∞	∞	∞	∞
t_1	0	10	1	∞	∞	∞	∞	∞	∞
t_2	0	10	1	2	∞	∞	11	∞	∞
t_3	0	7	1	2	3	∞	11	12	∞
t_4	0	4	1	2	3	4	8	12	13
t_5	0	4	1	2	3	4	5	9	5

Table 3.4: State of execution till t_5; highlighted rows indicate latest locally consistent snapshots.

Step	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
t_6	0	7	1	∞	∞	∞	5	9	5
t_6	0	7	1	2	∞	6	5	9	5
t_7	0	7	1	2	3	6	5	9	5
t_8	0	7	1	2	3	4	5	9	5

Table 3.5: Recovering vertices v_3, v_4 and v_5.

Next we illustrate the necessity of PR-Ordering during recovery using our SSSP example. Let us assume that c_0 and c_1 fail after t_5. The locally consistent snapshots captured at c_0 and c_1 are highlighted in Table 3.4. Thus, during recovery, the local state of c_2 is that at t_5 while the latest available snapshots s_0 and s_1 from c_0 and c_1 represent their execution states at t_3 and t_1 respectively. By examining the dependences in the computation, we easily determine that $s_1 \leq_{PR} s_0 \leq_{PR} c_2$. Therefore, s_1 is processed first using values from s_0 and c_2 resulting in values shown in Table 3.5. After t_8, recovery for s_0 can read from the computed results for v_3, v_4 and v_5 because s_0 is PR-Consistent with these values. Finally, as s_0 is PR-Consistent with c_2, processing occurs for values v_0 to v_5 alone, as shown in Table 3.6. After t_9, all the values are PR-Consistent with each other, i.e., recovery is complete and processing resumes from this state. Note that if we had ignored PR-Ordering monotonicity would be violated. For example, if we had first performed computation over s_0, then by reading $v_3 = \infty$ from s_1, v_4 would have been computed as 10, violating monotonicity as $10 > 7$.

Step	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
t_8	0	7	1	2	3	4	5	9	5
t_9	0	4	1	2	3	4	5	9	5

Table 3.6: Recovering vertices v_0 to v_5.

Theorem 3.2.2. *A total PR-Ordering of S is a necessary condition to recover from failures impacting machines in F, for all possible $F \subset C$, to a PR-Consistent state using a locally consistent snapshot S.*

Proof. We must show that if the execution state recovers to a PR-Consistent state using S, then a total PR-Ordering of S must be available. We prove this by contraposition. Let us assume that a total PR-Ordering of S is not available and hence, snapshots of failed machines $c_i, c_j \in F$, i.e., $s_i, s_j \in S$, are not ordered under $\leq_{PR}$. Without loss of generality, we focus on how the local execution state for c_i can be restored to e_i^r using s_i so that $E = E^{cf} \cup \{e_i^r\}$ is PR-Consistent. When e_i^r is initialized with s_i, E cannot be guaranteed to be PR-Consistent because $\forall e_k^c \in E^{cf}$, $e_k^c \leq_{PR} s_i$ cannot be guaranteed. Hence, e_i^r must be processed further after it is initialized using values from s_i. While processing e_i^r, values of boundary vertices from c_j can be either (Case 1) read from s_j or (Case 2) not read.

Case 1: Processing of e_i^r reads from s_j. In this case PR-Semantics cannot be guaranteed since $s_i \not\leq_{PR} s_j$ may be true. Hence, e_i^r represents an inconsistent state.

Case 2: Processing of e_i^r does not read from s_j. In this case after e_i^r is computed, $\forall e_k^c \in E^{cf}$, $e_k^c \leq_{PR} e_i^r$ cannot be guaranteed because e_k^c could have observed a fresher value from e_i^c prior to failure which was in turn calculated from e_j after s_j was captured. Moreover, recovery of local execution state for c_j cannot be initiated at this point due to the same choice of whether it could or could not read from e_i^r.

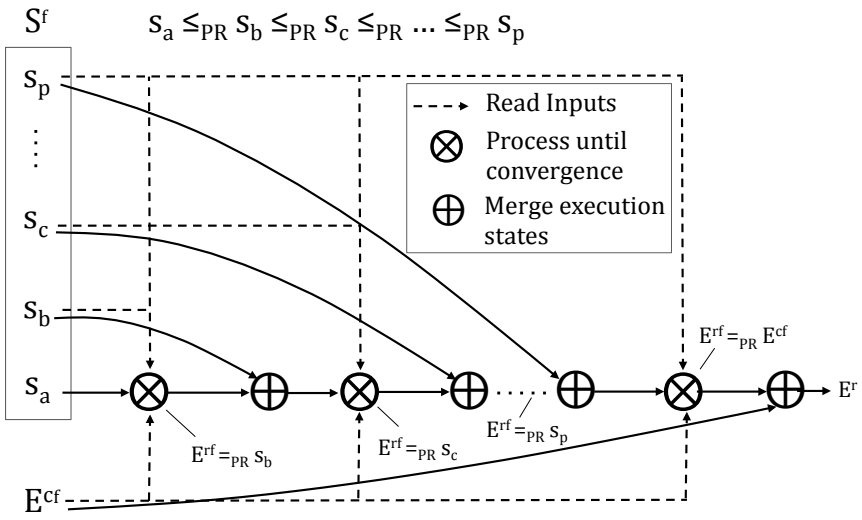


Figure 3.3: Recovery from multiple failures.

This means, e_i^r cannot be constructed such that PR-Consistency for $E = E^{cf} \cup \{e_i^r\}$ is guaranteed. Since failures can impact machines in any non-empty $F \in PowerSet(C)$, recovery of execution state of any failed machine such that it is PR-Consistent with e_k^c, $\forall e_k^c \in E^{cf}$ is not possible if total PR-Ordering of S is not available. □

Now that we know that the total PR-Ordering of S is required for recovery, we aim to construct the PR-Consistent state assuming that such a total PR-Ordering of S is available. The basic idea is to recover the execution states of individual machines one by one (see Figure 3.3), in an order such that PR-Semantics is never violated.

Algorithm 3 shows recovery from multiple failures. The PR-Consistent execution state (E^{rf}) is constructed by repetitively performing the *adding* and *forwarding* of the saved states from failed machines. The addition of execution states using saved snapshots is done in PR-Order (line 5) to guarantee that PR-Consistency is always retained in E^{rf}. During forwarding, the processing reads inputs, i.e., boundary vertices, from two sources (line 13): the snapshots from failed machines that are not yet incorporated in E^{rf} (i.e.,

Algorithm 3 Recovery from multiple failures.

```
1: S^f: Local snapshots captured on failed machines
2: E^cf: Execution state of remaining machines at the time of failure
3: function RECOVER ( )
4:     E^rf ← ∅
5:     P^f ← SORTASCENDING(S^f)
6:     while P^f ≠ ∅ do
7:       ▷ Adding Phase
8:         s_i ← GETFIRST(P^f)
9:         REMOVE(P^f, s_i)
10:        e_i^r ← LOADSUBGRAPH(s_i)
11:        E^rf ← E^rf ∪ {e_i^r}
12:      ▷ Forwarding Phase
13:        READBOUNDARYVERTICES(E^rf, P^f, E^cf)
14:        PROCESSUNTILCONVERGENCE(E^rf)
15:    end while
16:    E^r ← E^rf ∪ E^cf
17:    return E^r
18: end function
```

$\forall s_i \in P^f$), and from current execution states of remaining machines (i.e., $\forall e_i^c \in E^{cf}$). At the end of the while loop (lines 6-15), E^{rf} represents the workload of failed machines that is PR-Consistent with the current execution state of the remaining machines (E^{cf}) and hence, the two execution states are merged (line 16) to form the required E^r.

Theorem 3.2.3. *Algorithm 3 recovers a PR-Consistent execution state.*

Proof. We prove this by first showing that at any point in the algorithm, E^{rf} is PR-Consistent, and then proving that at the end, E^{rf} becomes PR-Consistent with E^{cf}, resulting in E^r to be PR-Consistent.

The algorithm uses $S^f = \{s_i \mid c_i \in F\}$; without loss of generality, let $s_a \leq_{PR} s_b \leq_{PR} s_c \leq_{PR} ... \leq_{PR} s_p$ be the total PR-Ordering of S^f. At each step, the algorithm adds a snapshot into the recovery process in this PR-Order (see Figure 3.3).

Initially, $E^{rf} = \varnothing$ and e_a^r is initialized with values from s_a. This e_a^r is added to E^{rf}. Due to the available PR-Ordering, e_a^r can read boundary vertices available from s_i, $\forall s_i \in S^f \setminus \{s_a\}$. Also, e_a^r can read from e_i^c, $\forall e_i^c \in E^{cf}$. Hence, processing E^{rf} by allowing it to read boundary vertices from these sources does not violate PR-Semantics.

Since the forwarding phase fully processes E^{rf} until convergence, e_a^r is now based on values from s_b and from other sources which were available even after s_b was captured. On the other hand, when s_b was captured, its values were based on reads from c_a which were not based on fresher values from other sources. Hence, $s_b \leq_{PR} e_a^r$ which further leads to $e_a^r =_{PR} s_b$. This means, when e_b^r is added to E^{rf}, E^{rf} is still PR-Consistent.

Again, e_b^r can read from s_i, $\forall s_i \in S^f \setminus \{s_a, s_b\}$, and also from e_i^c, $\forall e_i^c \in E^{cf}$ which allows processing of E^{rf} to read boundary vertices from these sources without violating PR-Semantics. After the forwarding phase, using the same argument as above, we can show that $e_a^r =_{PR} e_b^r =_{PR} s_c$ which allows e_c^r to be added to E^{rf} while ensuring E^{rf} remains PR-Consistent.

At every step of this construction process, $|E^{rf}|$ increases by 1. When $|E^{rf}| = |F|$, we achieve E^{rf} such that it is only based on values from E^{cf} and hence, $\forall e_i^r \in E^{rf}$ and $\forall e_j^c \in E^{cf}$, $e_j^c \leq_{PR} e_i^r$ which further leads to $E^{rf} =_{PR} E^{cf}$. Hence, the constructed $E^r = E^{rf} \cup E^{cf}$ is a PR-Consistent execution state. □

Maintaining PR-Ordering after Recovery. After the recovery process, $\forall s_i \in S, s_i \leq_{PR} E^{rf}$. Hence, the future snapshots captured after the recovery process are also PR-Ordered with snapshots in S. In case of any further failures, the available snapshots in S before the previous failure can be used along with the newly captured snapshots following recovery.

Thus, the snapshots in S and newly captured snapshots collectively guarantee that local states of all machines are available.

Cascading Failures. Failures can occur at any point in time during execution and hence, the recovery process can be affected by new failures at remaining machines. Such cascading failures need to be handled carefully so that the PR-Consistent state constructed by the recovery process includes the workload from newly failed machines.

Since Algorithm 3 incrementally constructs E^{rf} while maintaining the invariant that it is always PR-Consistent, the snapshots of newly failed nodes cannot be directly incorporated in the recovery process. This is because E^{rf} is processed based on values from E^{cf} and allowing a new snapshot to join the recovery process will cause older values to be read by E^{rf} thus violating PR-Semantics. Moreover, the new snapshots cannot be made PR-Consistent with E^{rf} since that in turn requires these snapshots to be PR-Consistent with E^{cf}. Hence, upon cascading failures, the recovery process discards the partially constructed E^{rf} and resumes the process by recreating the linear plan (P^f) consisting of all the failed nodes and then incrementally constructing the PR-Consistent execution state E^{rf}.

Machines Participating in Recovery. In a fail-stop failure model, the snapshots must be replicated on different machines so that they are available for recovery. There are two main ways to replicate a snapshot: either replicate it in entirety on a remote machine, or partition the snapshot into smaller chunks and distribute them across different machines. Both strategies have pros and cons. Placing the snapshot entirely on a single machine allows *confined* recovery for single machine failure with minimal communication. However,

this comes at the cost of workload imbalance during post-recovery processing. Partitioning the snapshot and scattering the chunks across the remaining machines provides better load balancing.

3.2.3 Capturing PR-Ordering

Since the PR-ordering captures the causality relationship across different machines, we use logical timestamps to enable ordering of snapshots. We rely on a light-weight centralized timestamp service to ensure that correct global ordering of logical timestamps is possible. The role of the timestamp service is to atomically provide monotonically increasing timestamps; this does not require synchronization between the machines, allowing asynchronous processing and checkpointing to continue concurrently.

The ordering is captured using a lightweight 3-phase protocol by ensuring that the local execution state to be checkpointed does not change with respect to any new remote input coming during the checkpointing process. The first phase is the *Prepare* phase that blocks the input stream representing remote reads, and then gets a logical timestamp for the snapshot from the distributed coordinator. The second phase is the *Snapshot* phase during which the execution state of the snapshot is actually captured. This phase overlaps computation over vertices while capturing the local snapshot by enforcing that vertex values are saved before they are updated (as in GraphLab [84]) which leads to a locally consistent snapshot (i.e., ensures [P-LCO]). Finally, the third phase is the *Resume* phase which marks the end of snapshot with the acquired logical timestamp and unblocks the input stream to allow future reads. Algorithm 4 summarizes the above protocol for performing local checkpointing which generates correct PR-Ordering across the captured snapshots. The

getNextLogicalTimeStamp() function atomically provides a monotonically increasing logical timestamp.

Algorithm 4 Local checkpointing algorithm.

```
1: function CHECKPOINT ( )
2:   ▷ Prepare Phase
3:     blockIncomingMessages( )
4:     t_s ← getNextLogicalTimeStamp( )
5:   ▷ Snapshot Phase
6:     snapshotUpdate( )
7:   ▷ Resume Phase
8:     save(END-CHECKPOINT, t_s)
9:     unblockIncomingMessages( )
10: end function
11:
12: function SNAPSHOTUPDATE ( )
13:     for v ∈ V do
14:       ▷ Snapshot Vertex
15:         if v is to be checkpointed then
16:             save(v)
17:         end if
18:       ▷ Process Vertex
19:         if v is to be processed then
20:             process(v)
21:         end if
22:     end for
23: end function
```

Theorem 3.2.4. *Algorithm 4 generates a correct total PR-Ordering of local snapshots ϵ S.*

Proof. We first show that the generated PR-Ordering is a total ordering, and then show its correctness.

Total Ordering: Each of the local snapshots captured is assigned a unique timestamp via the distributed coordinator. Hence, $\forall s_i, s_j \in S$, their timestamps, t_{s_i} and t_{s_j} are ordered, i.e., either $t_{s_i} < t_{s_j}$ or $t_{s_j} < t_{s_i}$. By mapping this timestamp ordering between t_{s_i} and t_{s_j}

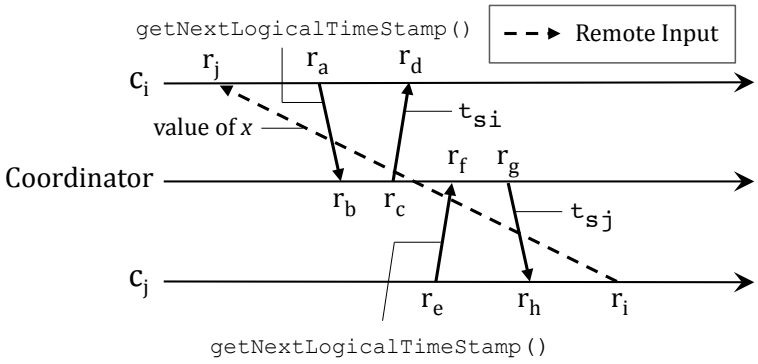


Figure 3.4: Event sequence with incorrect access of value x.

to the PR-Ordering between s_i and s_j, we achieve either $s_i \leq_{PR} s_j$ or $s_j \leq_{PR} s_i$. Since this mapping is done for every pair of snapshots in S, S is totally ordered under $\leq_{PR}$.

CORRECT PR-ORDERING: We prove this by contradiction. Let us assume that $s_i \leq_{PR} s_j$ is an incorrect PR-Ordering. This ordering is a result of mapping from timestamp relation $t_{s_i} < t_{s_j}$. Since the logical timestamps are monotonically increasing in the order of arrival of requests, the timestamp request from node c_i should have arrived before than that from node c_j in real time space.

Without loss of generality, Figure 3.4 shows the sequence of events representing our current case. Note that r_a through r_j indicate real time points in the global real time space. We know the following orderings are valid.

$$r_a < r_b \quad \text{(send-receive ordering)} \tag{3.1}$$

$$r_b < r_f \quad \text{(request arrival ordering)} \tag{3.2}$$

$$r_f < r_g \quad \text{(causality ordering)} \tag{3.3}$$

$$r_g < r_h \quad \text{(send-receive ordering)} \tag{3.4}$$

Moreover, since our assumption is that $s_i \not\leq_{PR} s_j$, there should be a value x which is read from c_j to c_i (indicated via dotted arrow) with the following ordering constraints:

$$r_j < r_a \quad \text{(prepare phase ordering)} \tag{3.5}$$

$$r_h < r_i \quad \text{(prepare phase ordering)} \tag{3.6}$$

$$r_i < r_j \quad \text{(send-receive ordering)} \tag{3.7}$$

Combining Equations 1-6 leads to $r_j < r_i$ which contradicts Equation 7. Hence, our assumption is false, i.e., $s_i \leq_{PR} s_j$ is a correct PR-Ordering. □

Theorem 3.2.5. *Algorithm 4 generates a strict total PR-Ordering across local snapshots in S, i.e., $\forall s_i, s_j \in S$, if $s_i \leq_{PR} s_j$, then $s_j \not\leq_{PR} s_i$.*

Proof. The $\leq_{PR}$ ordering is mapped from the ordering of logical timestamps assigned using GETNEXTLOGICALTIMESTAMP() function which atomically provides monotonically increasing timestamps. □

Theorem 3.2.5 indicates two things: first, the locally consistent checkpointing process generates local snapshots that are considered to be inconsistent with other local snapshots; even if the snapshots captured are truly globally consistent, the monotonic nature of timestamps assigned to snapshots does not capture this information. Secondly, the schedule for recovery from multiple failures is deterministic.

Missing Snapshots. Failures can occur even before the first set of snapshots from the affected machines are available. Recovery from such failures is done from the initial state of the

affected machine's workload. To ensure PR-Semantics is adhered to during the recovery process, a total PR-Ordering must be available across the initial states for different machines' workload and the captured snapshots. Such PR-Ordering is available naturally by viewing the initial states to have read no values from other workloads. Let $I = \{i_0, i_1, ..., i_{k-1}\}$ represent the set of initial local states of machines in cluster C.

Corollary 3.2.2. *$\forall i_i \in I$ and $\forall s_j \in S$, $i_i \leq_{PR} s_j$.*

Moreover, the total PR-Ordering among the individual initial states can be captured as follows.

Corollary 3.2.3. *$\forall i_i, i_j \in I$, $i_i \leq_{PR} i_j$ and $i_j \leq_{PR} i_i$. This means, $i_i =_{PR} i_j$.*

Corollary 3.2.3 captures the PR-equivalence across initial states which means, processing an initial state using values from other initial states adheres to PR-Semantics. For simplicity, we consider the initial states as snapshots captured at the beginning of processing and assume the PR-Ordering based on the ordering of machine ids, i.e., if $\forall c_i, c_j \in C$, if $i < j$ then $i_i \leq_{PR} i_j$.

			Checkpoint Frequency (sec)			
Graphs	**#Edges**	**#Vertices**	**PR**	**MSSP**	**CC**	**KC**
Twitter (TT) [72]	1.5B	41.7M	200	30	100	200 ($k = 10$)
UKDoman (UK) [11]	1.0B	39.5M	100	30	100	50 ($k = 20$)
LiveJournal (LJ) [5]	69M	4.8M	10	2	1	10 ($k = 50$)

Table 3.7: Real world input graphs and benchmarks used.

	LJ	UK	TT
PR	22.8	301.6	474.9
MSSP	4.1	54.9	50
CC	4.2	102.2	78.3
KC	30.8	162.7	364.5

Table 3.8: Execution times (sec).

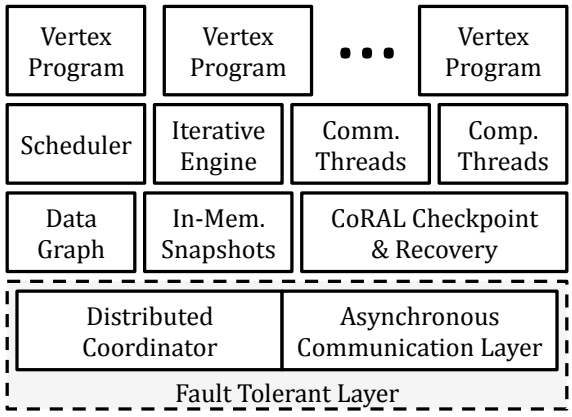


Figure 3.5: System design.

3.3 Evaluation

System Design. We incorporated CoRAL in an asynchronous iterative graph processing system based on ASPIRE [129] as shown in Figure 3.5. A fault tolerant layer is designed to handle distributed coordination across multiple machines and provide asynchronous communication. The distributed coordinator is based upon Apache Zookeeper [55]. It manages membership of machines, detects machine failures, and invokes callbacks to CoRAL module. It also provides atomic timestamp service required for capturing PR-Ordering, and synchronization primitives like barriers for programmability. The asynchronous communication layer is built using non-blocking primitives provided by ZeroMQ [146].

The application layer includes the graph processing engine which operates on the given graph. The graph is partitioned using GraphLab's partitioner that greedily minimizes

the edge-cut. The CoRAL checkpointing and recovery module periodically captures locally consistent snapshots of the graph state and ensures total ordering by coordinating with Zookeeper. Upon failures, it recovers the lost graph state using confined recovery.

Experimental Setup. Our evaluation uses four algorithms: PageRank (PR) [94], MultipleSourceShortestPaths (MSSP), ConnectedComponents (CC) [152], and KCoreDecomposition (KC) taken from different sources [84, 129]. The algorithms are oblivious to the underlying fault tolerance mechanisms used in our evaluation and hence, no modifications were done to their implementations. They were evaluated using real-world graphs listed in Table 5.2 and running them until convergence. The k parameter for KC is also listed.

To evaluate the effectiveness of locally consistent checkpointing and recovery mechanism, we set the checkpointing frequency in our experiments such that 3-6 snapshots are captured over the entire execution lifetime. The checkpoint frequencies used in our evaluation are shown in Table 5.2. While capturing locally consistent snapshots allows relaxing the time at which different local checkpoints can be saved, for KC we limit this relaxation to within the same k so the snapshot is fully captured within the same engine invocation.

All experiments were conducted on a 16-node cluster on Amazon EC2. Each node has 8 cores, 64GB main memory, and runs 64-bit Ubuntu 14.04 kernel 3.13.

Techniques Compared. We evaluate CoRAL using ASPIRE distributed asynchronous processing framework. ASPIRE guarantees PR-Semantics and it performs well compared to other frameworks as shown in [129]. For comparison with other systems, the raw execution times (in seconds) for ASPIRE are shown in Table 3.8.

Our experiments compare two fault tolerant versions that are as follows: **CoRAL** captures locally consistent snapshots and performs confined recovery to PR-Consistent state; and **BL** is the baseline technique used by asynchronous frameworks like GraphLab [84]. It captures globally consistent snapshots based on the the Chandy-Lamport snapshot algorithm [19] and recovers by rolling back all machines to the most recent checkpoint.

To ensure correct comparison between the two versions, failures are injected to bring down the same set of machines when same amount of progress has been achieved by the iterative algorithms. We check the execution state that is present immediately after failure to confirm that the vertex values are essentially the same (within tolerable bound for floating point values) for BL and CoRAL so that recovery starts from the same point. We also evaluate our recovery mechanism by starting the program assuming that failure has already occurred; we do this by feeding the same execution state (vertex values) as initializations and starting the recovery process; the performance results are same in this case too. CoRAL guarantees correctness of results; thus, final results of BL and CoRAL for each experiment are 100% accurate.

3.3.1 Recovery Overhead

Single Failure. We measured the execution times of CoRAL and BL when a single failure occurs during the program run. The execution times after the occurrence of failures (i.e., recovery and post recovery), normalized with respect to execution time for BL, are shown in Figure 3.6. The complete execution times (in seconds) including the execution prior to failure are given in Table 3.9.

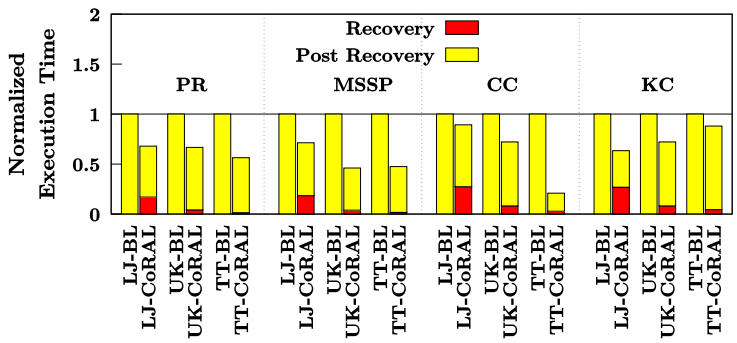


Figure 3.6: CoRAL vs. BL: Single failure execution times normalized w.r.t. BL.

		LJ	UK	TT
PR	**BL**	31.24	334.78	603.12
	CoRAL	24.85	322.71	398.92
MSSP	**BL**	8.73	69.72	57.00
	CoRAL	6.19	53.25	40.50
CC	**BL**	6.80	121.62	173.04
	CoRAL	6.80	102.20	84.56
KC	**BL**	64.68	195.24	612.36
	CoRAL	44.35	157.82	539.46

Table 3.9: CoRAL vs. BL execution times (sec) for single machine failure.

We observe that CoRAL quickly recovers and performs faster compared to BL in all cases – on an average across inputs, CoRAL is 1.6×, 1.7×, 1.3× and 2.3× faster than BL for PR, MSSP, CC, and KC respectively. We also found that the recovery process of CoRAL is lightweight – on an average across benchmarks, the recovery process takes 22.5%, 3.5%, and 3.3% of the total execution time for inputs LJ, UK, and TT. More importantly, the percentage time taken by the recovery process reduces as graph size increases.

Furthermore, in some cases we observed that for CoRAL, the overall execution time starting from the beginning of the iterative processing goes below the original execution time – for example, for both PR and MSSP on TT, the overall execution time reduces by 15.7% and 18.8%. This is because the CoRAL recovery constructs a PR-Consistent state with fresher values that is closer to the final solution, so the convergence is achieved faster.

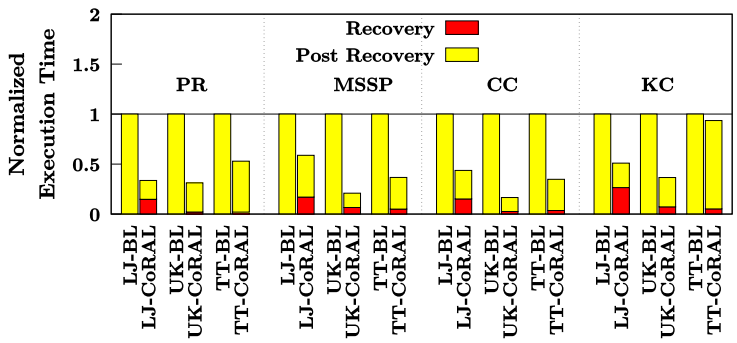


Figure 3.7: CoRAL vs. BL: Recovery for single failure from initial state. Execution times normalized w.r.t. BL.

The preceding experiment showed the combined benefit of lightweight checkpointing and confined recovery. Next we conducted an experiment to determine the benefit of confined recovery alone. We turned off checkpointing and upon (single) failure rolled back execution to initial default values – CoRAL only rolls back state of failed machine while BL must roll back states of all machines. Figure 3.7 shows that on an average across inputs, CoRAL executes for only 0.4×, 0.4×, 0.3× and 0.6× compared to BL for PR, MSSP, CC, and KC respectively. While BL recovery is fast as it simply reverts back to the initial state, the computation performed gets discarded and much time is spent on performing redundant computations. Moreover, we observed an increase in the overall execution time (starting from the beginning of processing) for BL by 1.1–1.9× which is due to the same amount of work being performed by fewer machines after failure. CoRAL, on the other hand, does not discard the entire global state, and hence finishes sooner.

Multiple Failures. Figure 3.8 shows the performance for multiple failures for PR benchmark on UK graph. We simultaneously caused failure of 1 through 6 machines to see the impact of our strategy. As we can see, CoRAL performs 1.5–3.2× faster than BL. We

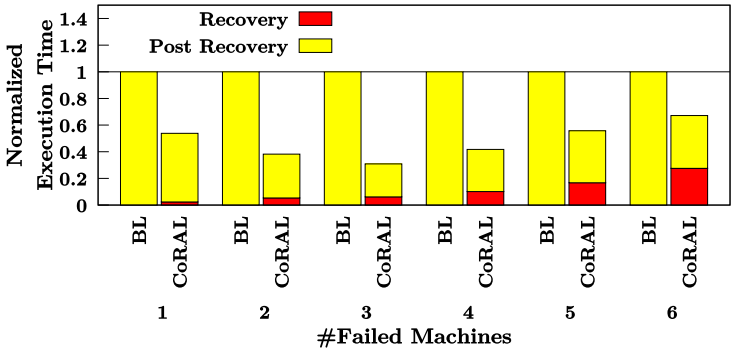


Figure 3.8: CoRAL vs. BL: Varying number (1 to 6) of machine failures. Execution times for PR on UK normalized w.r.t. BL.

observed that the overall execution times for BL after occurrence of failures increase up to 1.2×, whereas CoRAL takes only 0.2–0.8× of time to recover and finish processing. This is because CoRAL does not discard the progress of machines that are not impacted by failure.

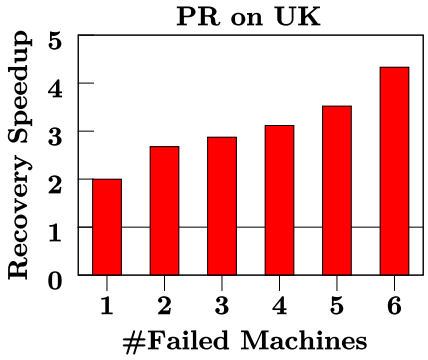
Note that the execution times increase as the number of simultaneously failing machines increase. This is due to two reasons. First, the remaining (non-failed) state becomes smaller and the lost states become larger, causing more work during recovery and post-failure. Second, after failure, the processing continues on fewer leftover machines, i.e. computation resources decrease. It is also interesting to note that CoRAL's recovery time also increases with increase in the number of simultaneously failed machines due to the linear nature of our recovery strategy.

3.3.2 Partitioning Snapshots: Impact on Recovery

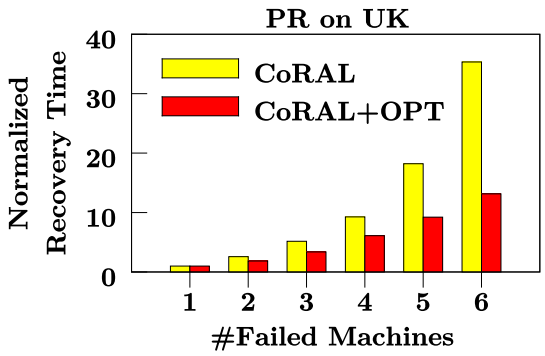
During checkpointing, a local snapshot can be saved by partitioning them and placing individual chunks on different machines, or by placing the entire snapshot on a single machine. Based upon the manner in which snapshots are saved, only the machines on

which snapshots are locally available can quickly perform recovery. While partitioning the snapshot allows more machines to participate in the recovery process, placing the snapshots without partitioning reduces the communication during the recovery process – for example, for a single machine failure, communication is not required during the recovery process where PR-Consistent state is constructed by iteratively processing until convergence.

Figure 3.9a evaluates this design choice by showing the speedups achieved during recovery using the partitioning strategy over maintaining the snapshot as a whole. The results show that allowing multiple machines to take part during recovery process overshadows the communication increase and accelerates recovery. Also the speedups are higher when greater number of machines fail. While not partitioning leads to an increase in the recovery workload by only a constant factor (i.e., size of a single snapshot), when more machines fail, the communication required to process the workload increases which limits the speedups.



(a) Speedup in CoRAL recovery due to partitioning snapshots.



(b) Recovery time with and without optimization normalized w.r.t. single failure case.

Figure 3.9

3.3.3 Optimizing Recovery from Multiple Failures

Various works suggest users often disable checkpointing (GraphX [45], Giraph [25], and Distributed GraphLab [84]) to eliminate its overheads. The PR-Consistent state can be constructed even when no snapshots are captured using initial state, i.e., default values. Moreover, Corollary 3.2.3 suggests that the execution state using default values is already PR-Consistent. Hence, the recovery process can be further optimized to incorporate the states of all the failed machines together, instead of adding them one by one. When the entire failed state is fully processed, it becomes PR-Consistent with the available current execution states of the machines not impacted by failure. Hence, computation can resume using this PR-Consistent state.

Figure 3.9b shows the time taken by the CoRAL recovery using initial state, with and without the above optimization. The optimization further speeds up the recovery process by an order of magnitude. This observation can be incorporated in the checkpointing strategy itself – if checkpointing guarantees subsets of local snapshots to be PR-Consistent, the snapshots in those subsets can be incorporated together during the recovery process, instead of adding them one by one.

3.3.4 Checkpointing: Impact on Network Bandwidth

We now evaluate the benefits of using locally consistent checkpointing. During checkpointing, the captured snapshots are saved remotely so that they are available upon failure. This leads to an increase in network usage. In Figure 3.10, we measure the 99th

percentile [1] network bandwidth for BL and CoRAL by varying the replication factor (RF) from 1 to 6, normalized w.r.t. no replication [2].

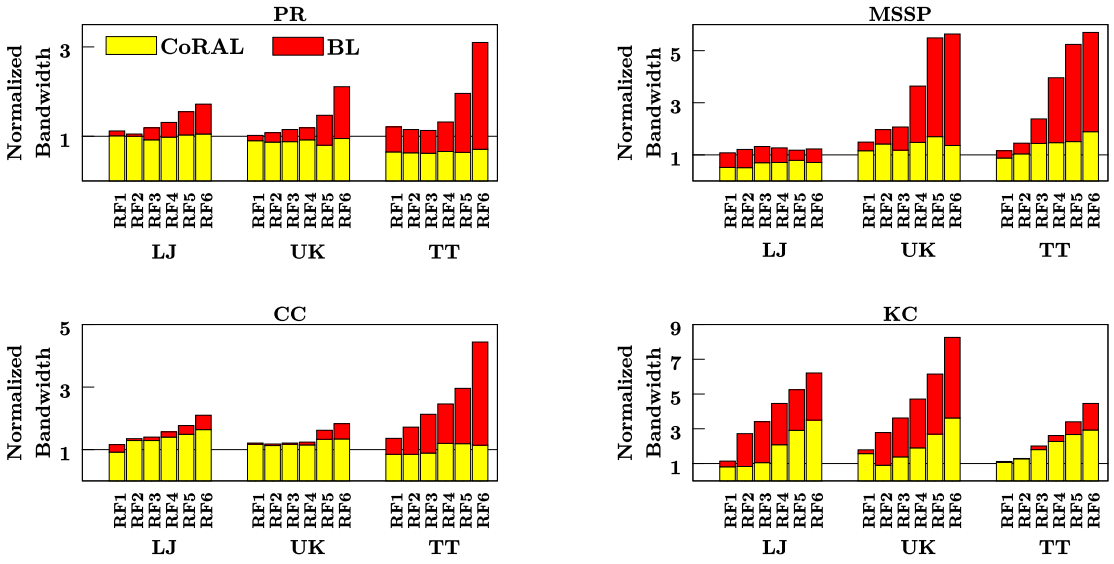


Figure 3.10: BL vs. CoRAL: 99th percentile network bandwidth for varying RF (1 to 6) normalized w.r.t. no checkpointing case.

As we can see, the peak bandwidth consumption increases rapidly with increase in RF for BL because the consistent checkpointing process saves all the snapshots at the same time, which leads to simultaneous bulk network transfers. The peak bandwidth consumption for CoRAL does not increase as rapidly – this is because CoRAL staggers the capturing of different snapshots over time, and hence, the snapshots are transferred to remote machines at different points in time at which they become available. On an average across all benchmark-input configurations, there is a 22% to 51% reduction in 99th percentile bandwidth using CoRAL as RF is varied from 1 to 6.

[1] The performance trend is similar for higher percentile values.
[2] The network statistics were measured using `tcpdump`.

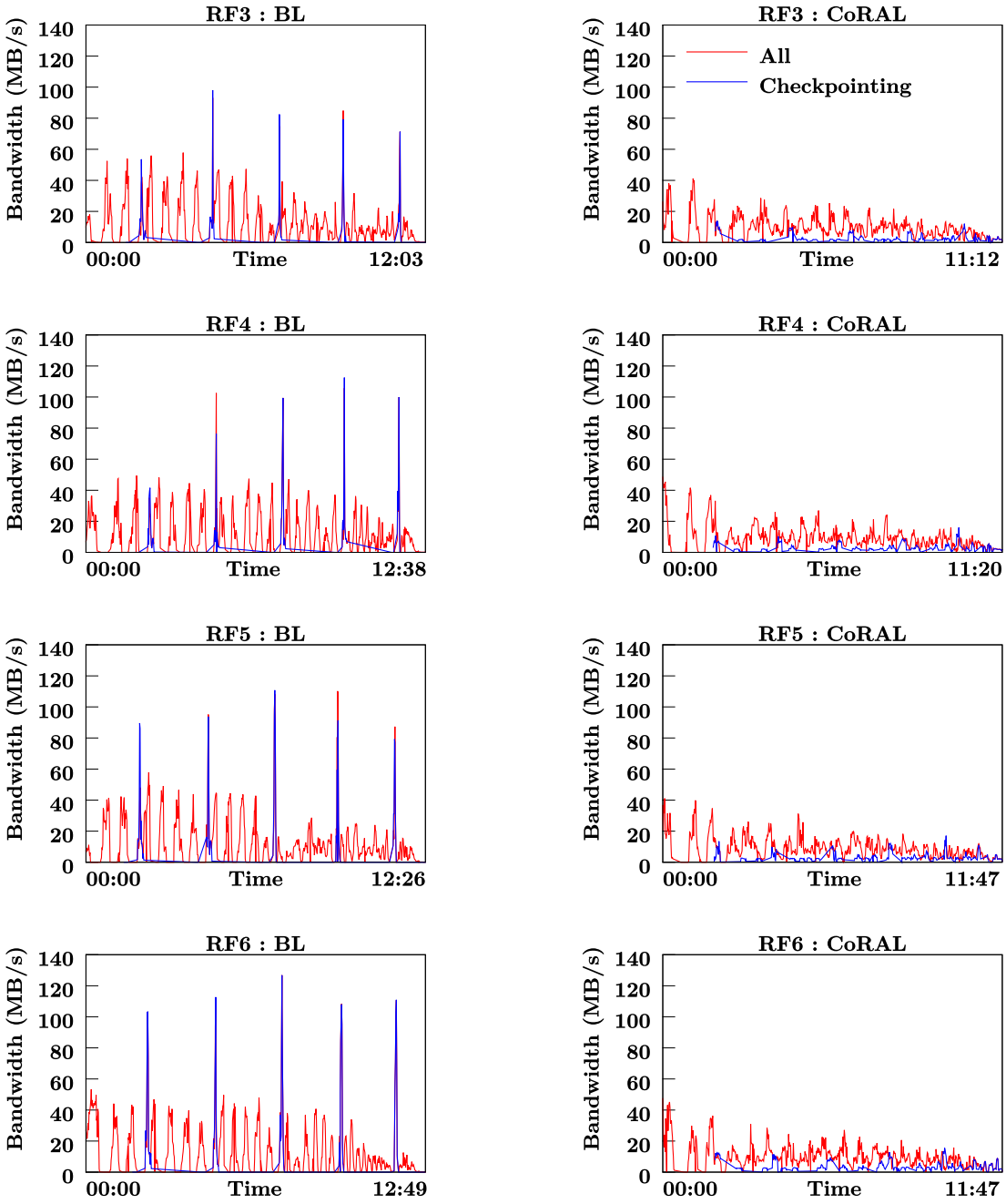


Figure 3.11: BL vs. CoRAL: Network usage for PR on UK.

There is a noticeable increasing trend for CoRAL on KC – this is mainly because the checkpointing process in KC can be relaxed only during computation for a given core,

which in certain cases became a rather narrow window over which all the snapshots had to be transferred.

Figure 3.11 shows the bandwidth consumption for PR on UK with RF varying from 3 to 6. Here All (red lines) indicate total bandwidth usage while Checkpointing (blue lines) indicate bandwidth consumed due to checkpointing alone. As we can see, for BL, the bandwidth periodically increases; moreover, the intermittent spikes are due to the checkpointing process. This is mainly because the local snapshots from all machines are sent and received at the same time. For CoRAL, the checkpointing process on different machines can take place at different times and hence, the transfer of local snapshots is spread over time, reducing the effect of bulk transfer and reducing network contention.

3.4 Summary

In this chapter, we studied the semantics of asynchronous distributed graph processing that enable supporting fault tolerance at reduced costs. We further discussed how confined recovery is achieved following failures by constructing alternate PR-Consistent state without discarding any useful work performed on non-failing machines. CoRAL uses locally consistent snapshots that are captured at reduced peak network bandwidth usage for transferring snapshots to remote machines. Our experiments confirmed reductions in checkpointing and recovery overhead, and low peak network bandwidth usage.

So far in this thesis we developed techniques to improve processing of graphs whose structure does not change. In the next chapter, we will see how asynchrony can be used to improve processing of dynamic graphs.

Chapter 4

Evolving Graph Processing

So far in this thesis, we developed techniques to improve processing of graphs whose structure does not change; however, an important feature of real-world graphs is that they are constantly evolving (e.g., social networks, networks modeling the spreading of diseases, etc.) [101, 136]. Such dynamic graphs are useful to capture dynamic properties and interesting trends that change over time. In this chapter, we exploit the asynchronous nature of graph algorithms to efficiently process *evolving graphs* where details about the graph evolution are available to perform temporal graph analyses. In particular, we develop strategies for *computation reordering* and *incremental processing* to reduce the overall communication and computation costs incurred during evolving graph processing.

The analysis of an evolving graph is expressed as the *repeating* of graph analysis over multiple snapshots of a changing graph – different snapshots are analyzed independently of each other and their results are finally aggregated. Note that evolving graph processing is different from streaming graph processing (Chapter 5) where iterative processing

continues over a dynamic graph structure that keeps on changing, hence terminating to the final solution for the most updated graph.

Due to the fast-changing nature of a modern evolving graph, the graph often has a large number of snapshots; analyzing one snapshot at a time can be extremely slow even when done in parallel, especially when these snapshots are large graphs themselves. For instance, one single snapshot of the Twitter graph [17] has over 1 billion edges, and there are in all 25.5 billion edges in all its snapshots we analyzed.

In this chapter, we develop temporal execution techniques that significantly improve the performance of evolving graph analysis, based on an important observation that different snapshots of a graph often have large overlap of vertices and edges. By laying out the evolving graph in a manner such that this temporal overlap is exposed, we identify two key optimizations that aid the overall processing: first, we reorder the computations based on loop transformation techniques to amortize the cost of fetch across multiple snapshots while processing the evolving graphs; and, second we enable feeding of values computed by earlier snapshots into later snapshots to amortize the cost of processing vertices across multiple snapshots. Furthermore, the two optimizations are orthogonal i.e., they amortize different costs, and hence, we identify and exploit the synergy between them by allowing feeding of values from all vertices, including those that haven't attained their final values, to amortize the processing cost, while simultaneously reordering computations to amortize the fetch cost.

Our optimizations are general and can be plugged into a variety of distributed or shared-memory graph processing systems. We incorporated our temporal execution tech-

niques in GraphLab [85] and ASPIRE [129] to add support for evolving graph processing in these frameworks. Our experiments with multiple real evolving graphs and graph processing algorithms on a 16-node cluster demonstrate that, on average fetch amortization speeds up the execution of GraphLab and ASPIRE by 5.2× and 4.1× respectively. Amortizing the processing cost yields additional average speedups of 2× and 7.9× respectively.

The rest of this chapter is organized as follows. Section 4.1 discusses the evolving graph and its iterative processing details. Section 4.2 discusses the space-efficient representation for evolving graphs. Section 4.3 and Section 4.4 discuss the optimizations to amortize the communication and computation costs incurred in evolving graph processing. Section 4.5 discusses how the amortization techniques are incorporated in different frameworks and Section 4.6 presents the experimental setup and the result analysis.

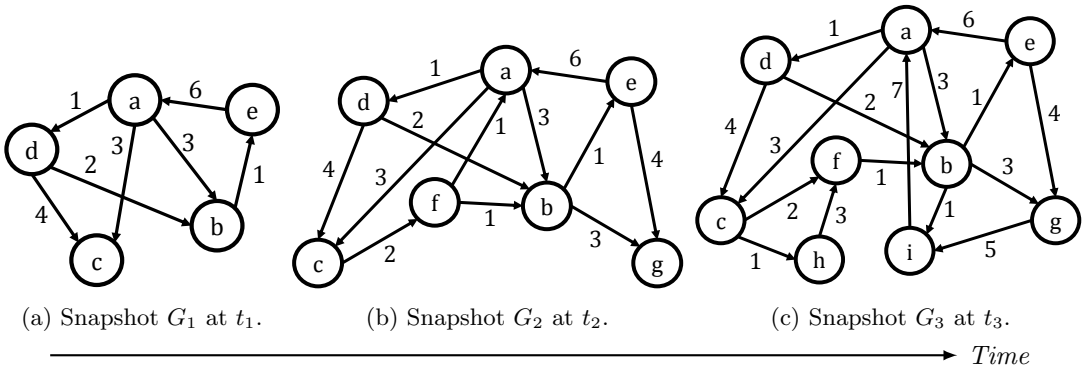
4.1 Evolving Graph and Iterative Processing

In this section, we first formalize evolving graphs and then discuss how they are iteratively processed. We will develop the temporal amortization techniques in subsequent sections using the processing details introduced in this section.

4.1.1 Evolving Graph

An evolving graph $\mathcal{G}$ is a graph that undergoes structural changes over time. These structural changes take place via *addition and deletion of edges and vertices*. Formally, an evolving graph $\mathcal{G} = \langle G_1, G_2, ..., G_k \rangle$ is a sequence of k *graph snapshots* taken at different points in time. In general the structural changes that cause a transition from snapshot

G_{i-1} to snapshot G_i involve both addition and deletion of edges and vertices. Note that a change in edge weight can be viewed as a deletion of the edge followed by its insertion with a different weight. Figure 4.1a, Figure 4.1b, and Figure 4.1c together show an evolving graph consisting of three graph snapshots taken at t_1, t_2, and t_3 respectively.



(a) Snapshot G_1 at t_1. (b) Snapshot G_2 at t_2. (c) Snapshot G_3 at t_3.

Figure 4.1: Example evolving graph $\mathcal{G} = \langle G_1, G_2, G_3 \rangle$.

4.1.2 Computation over Evolving Graphs

We briefly discuss the iterative vertex centric processing over a simple graph and then describe how it is performed over an evolving graph.

Iterative Vertex Centric Processing

In this work we focus on iterative vertex-centric graph algorithms, used in a wide range of modern mining and analytics tasks. In a vertex centric graph algorithm, computation is written from the perspective of a single vertex. For a given vertex, all the neighboring vertex values are fetched and a new value is computed using these fetched values. If the value of a vertex changes, its neighbors become *active*, i.e., they are scheduled to be processed in the next iteration. This process terminates when all the vertices in the graph become inactive.

In a given iteration, vertices are processed in parallel such that a vertex is completely processed by the same set of threads on the same machine. This results in a simple and intuitive parallel algorithm shown in function `EXECUTE()` in Algorithm 5. Since the techniques presented in this work are general and independent of any specific graph processing environment, we present the execution plan using high-level load/store primitives – fetching/storing data from/to (local or remote) machines is achieved transparently using the `FETCH/STORE` operations (lines 3, 6 and 14), which can have different implementations on different platforms; for example, a message-passing based system like Pregel [86] may use `SEND/RECEIVE` to transfer values across machines. Since only those vertices which are scheduled to be processed in a given iteration must be computed, `GET-ACTIVE-VERTEX` (line 2) returns the next vertex to be processed in the current iteration. While processing this vertex, the neighboring values are obtained using `GET-NEIGHBORS` (line 5) which internally fetches the neighboring vertex values residing on local and remote machines. Upon computation, if the change in the value of a vertex exceeds threshold ϵ (line 11), the neighbors of the vertex are activated for future processing by `ACTIVATE-NEIGHBORS` (line 12). It is interesting to note that the computations performed (line 9) are typically quite simple and threads are often seen waiting for values to be fetched (from local/remote memory/disk). In other words, these iterative graph algorithms are typically network-bound when executed on a distributed environment.

Algorithm 5 Iterative Algorithm on an Evolving Graph.

```
1:  function EXECUTE(G_i)
2:      for vid ∈ GET-ACTIVE-VERTEX(G_i) do
3:          vertex ← FETCH(vid, G_i)
4:          nbrs ← ∅
5:          for nid ∈ GET-NEIGHBORS(vertex) do
6:              nbrs ← nbrs ∪ FETCH(nid, G_i)
7:          end for
8:          old-value ← vertex.GET-VALUE( )
9:          comp-value ← f (vertex, nbrs)
10:         vertex.SET-VALUE(comp-value)
11:         if |old-value − comp-value| > ϵ then
12:             ACTIVATE-NEIGHBORS(vertex)
13:         end if
14:         STORE(vertex, G_i)
15:     end for
16: end function

17: function MAIN(𝒢)
18:     for each graph G_i ∈ 𝒢 do
19:         INITIALIZE(G_i)
20:         ACTIVATE-VERTICES(G_i)
21:         do
22:             parallel-for all threads do
23:                 EXECUTE(G_i)
24:             end parallel-for
25:             barrier( )
            /* Global termination condition */
26:         while there are active vertices
27:         OUTPUT-VERTEX-VALUES(G_i)
28:     end for
29: end function
```

Evolving Graph Processing

Analyzing an evolving graph involves *repeating* the iterative graph analysis computation over each graph snapshot. The `MAIN()` function in Algorithm 5 shows how an evolving graph is processed by invoking the same iterative `EXECUTE()` (line 23) function over different graph snapshots, one at a time. Since the results of iterative analysis are required for each of the graph snapshots, `OUTPUT-VERTEX-VALUES()` (line 27) is invoked immediately after processing of a given graph snapshot terminates.

Step	Vertex Distances					Active Vertices
	a	b	c	d	e	
0	∞	∞	∞	0	∞	-
1	∞	2	4	0	∞	b,c
2	∞	2	4	0	3	e
3	9	2	4	0	3	a
4	9	2	4	0	3	b,c,d

Table 4.1: Execution of SSSP on snapshot G_1 (Figure 4.1a).

Step	Vertex Distances							Active Vertices
	a	b	c	d	e	f	g	
0	∞	∞	∞	0	∞	∞	∞	-
1	∞	2	4	0	∞	∞	∞	b,c
2	∞	2	4	0	3	6	5	e,f,g
3	7	2	4	0	3	6	5	a,b,g
4	7	2	4	0	3	6	5	b,c,d

Table 4.2: Execution of SSSP on snapshot G_2 (Figure 4.1b).

Step	Vertex Distances									Active Vertices
	a	b	c	d	e	f	g	h	i	
0	∞	∞	∞	0	∞	∞	∞	∞	∞	-
1	∞	2	4	0	∞	∞	∞	∞	∞	b,c
2	∞	2	4	0	3	6	5	5	3	e,f,g,h,i
3	9	2	4	0	3	6	5	5	3	a,b,f,g,i
4	9	2	4	0	3	6	5	5	3	b,c,d

Table 4.3: Execution of SSSP on snapshot G_3 (Figure 4.1c).

As an example, the iterative Single Source Shortest Path (SSSP) computation for evolving graph snaphots G_1, G_2, and G_3 are shown in Tables 4.1, 4.2, and 4.3 respectively. Computation on G_1 proceeds as follows. Initially, all the vertices are set to ∞ and the source vertex (vertex d) is set to 0. The outgoing neighbors of d are b and c which are active in iteration 1. Their values are computed by fetching them and their incoming neighbors, i.e., vertices a, b, c and d. Since both b and c change, their outgoing neighbors (vertex e) become active in iteration 2. Processing terminates when there are no more active vertices. After processing completes for G_1, snapshot G_2 and later, snapshot G_3 is processed in similar manner (Table 4.2 and Table 4.3).

Note that evolving graph processing is inherently different from streaming graph processing [35]; streaming graph processing does not rely on strict notion of graph snapshots and iterative processing continues while the graph structure changes rapidly, hence terminating to the final solution for the most updated graph. Evolving graph processing, on the other hand, involves processing all the intermediate graph snapshots and producing the final converged results for each of the snapshots.

Since graphs being processed are large, significant effort is involved both in fetching of values into memory of processing sites and carrying out the required computation.

We recognize two opportunities (Section 4.3 and Section 4.4) to amortize these costs across multiple graph snapshots of an evolving graph. Note that the opportunities and proposed techniques are independent of any graph processing framework and any processing environment (distributed, shared-memory, out-of-core, etc.).

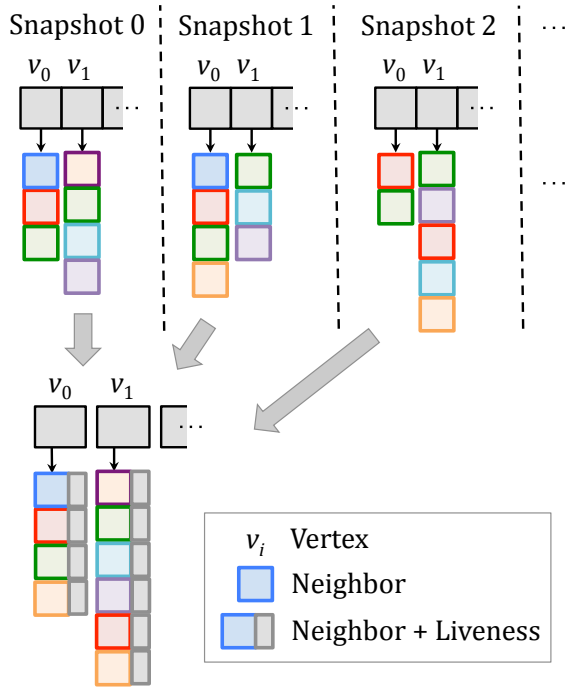


Figure 4.2: Temporal Layout of Evolving Graphs

4.2 Temporal Layout of Evolving Graphs

Computations performed over a static graph are typically vertex centric and hence, computation of a given vertex requires values of its neighboring vertices. To preserve locality in this case, graph processing systems typically lay out the graph structure using a variant of adjacency list format. Such a representation can be directly used to represent evolving graphs by laying out individual graph snapshots one after the other so that locality within each of the graph snapshot is preserved.

However, consecutive graph snapshots have high structural overlap which can be exploited to make the representation space efficient. Moreover, this structural overlap across snapshots can be used to expose the underlying temporal locality which can be further used to accelerate processing. Hence, we extend the traditional adjacency list based representation so that the represented graph structure is a union of all graph snapshots. This is analogous to using *structure-of-arrays (SoA)* layout instead of *array-of-structures (AoS)* layout where the graph snapshots collectively represent an array.

Figure 4.2 shows how we represent evolving graphs. The overall structure holds the union of all the captured graph snapshots while the information about the individual snapshots is maintained using liveness intervals tagged over edges. A liveness interval is represented by a *begin* and *end* timestamp and each edge in the evolving graph has a vector of such liveness intervals which allows tracking multiple additions and deletions of same edges throughout the lifetime of the application. These vectors are sorted based on the begin timestamps to allow quick access of required data using a *forward-only pointer*. Note that vertices do not need liveness intervals because the presence of a vertex in a snapshot can be determined by presence of its edges.

Every graph snapshot has an associated timestamp at which it was captured, i.e., it is the time up to which all the updates have been applied to the snapshot. An edge is considered part of the snapshot if the snapshot's timestamp falls in between the begin and end timestamps for some pair in the edge's liveness vector. Every edge also has a vector of immutable values to capture different edge-weights for different lifetimes.

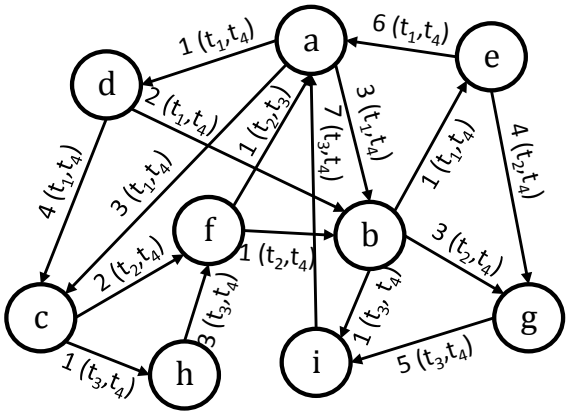


Figure 4.3: Evolving Graph $\mathcal{G}$ for example evolving graph in Figure 4.1.

Figure 4.3 shows an example of evolving graph $\mathcal{G} = \langle G_1, G_2, G_3 \rangle$ with 3 graph snapshots that are shown in Figure 4.1. The individual snapshots G_1, G_2, and G_3 taken at t_1, t_2, and t_3 are shown in Figure 4.1a, 2b and 2c respectively.

Space Complexity

An evolving graph with k snapshots, $\mathcal{G} = \langle G_1, G_2, ..., G_k \rangle$ consumes $\mathrm{O}\big(k \times (|V| + |E|)\big)$ space when it is represented as a sequence of separate snapshots. On the other hand, our unified representation consumes $\mathrm{O}\big(|V| + (p \times |E|)\big)$ space, where p is the maximum number of times any edge in the graph is updated. Typically, $p \ll k$ and hence, our unified representation is space efficient.

Compactly representing evolving graphs using lifetime intervals naturally exposes the temporal locality present across graph snapshots which can be exploited by changing the order in which processing occurs, as discussed in subsequent sections.

Step	Fetched Vertices		
	G_1	G_2	G_3
1	a,b,c,d	a,b,c,d,f	a,b,c,d,f
2	b,e	b,c,e,f,g	b,c,e,f,g,h,i
3	a,e	a,b,d,e,f,g	a,b,d,e,f,g,h,i
4	a,b,c,d	a,b,c,d,f	a,b,c,d,f

Table 4.4: Fetched Vertices for SSSP on G_1, G_2, and G_3.

Evolving Graph	Overlap
Twitter-25	66.0%
Delicious-100	58.8 %
DBLP-100	42.4 %
Amazon-100	44.8 %
StackEx-100	53.5 %
Epinions-100	51.2 %
Slashdot-70	47.2 %

Table 4.5: Average % fetch overlap across consecutive snapshots over different datasets (names include number of snapshots) for SSSP algorithm.

4.3 Fetch Amortization

The high structural overlap across multiple snapshots leads to similar patterns in fetch requests for vertices when the individual snapshots are processed. Table 4.4 provides the lists of *Fetched Vertices* for each iteration during the SSSP computation performed on three graph snapshots in Section 4.1. On comparing the *Fetched Vertices* columns for the corresponding (same) iterations of consecutive graph snapshots, we observe that a significant number of vertices that are fetched are common.

The high degree of fetch overlap can also be observed in real datasets. Table 4.5 presents the degree of fetch overlap across consecutive snapshots when computing SSSP for real world evolving graphs (see Table 4.8 in Section 4.6.1 for description of graphs). As we can see the fetch overlap is very high – over 49% of vertices are common among consecutive snapshots. This naturally leads us to the following question:

Can we re-order computations such that overlapping fetches can be batched together?

4.3.1 Fetch Amortization via Computation Re-ordering

The temporal layout of evolving graph which exposes the high structural overlap across multiple snapshots can be efficiently utilized to aggregate the fetch requests of same vertices across different snapshots. We achieve this by computation re-ordering enabled using loop transformation techniques.

In the original processing model (Algorithm 5), the outer `for` loop (line 18) processes graph snapshots one after the other while the inner `do-while` loop (line 21) processes a single snapshot. These two loops can be transformed such that the inner loop processes a batch of snapshots at the same time while the outer loop iterates through different batches of snapshots to be processed. Now, while simultaneously processing the batch of snapshots, the fetch requests for different versions of the same vertex can be aggregated together by fusing multiple versions of inner-most loop which processes active vertices (line 2) such that processing of multiple versions of the same vertex belonging to different snapshots occur at the same time. This allows aggregating the fetch requests performed for multiple versions of same vertices.

The above idea is generalized by simultaneously processing Δ consecutive graph snapshots in parallel.

Fetch Amortization (FA). Fetch amortization *simultaneously processes* Δ snapshots $(G_{i+1}, G_{i+2} \ldots G_{i+\Delta})$ so that fetches of vertices common among some of the Δ snapshots can be *aggregated* to amortize fetch cost across snapshots.

Algorithm 6 shows the iterative parallel algorithm that incorporates fetch amortization. Note that this algorithm processes Δ snapshots in parallel denoted by $\mathcal{G}\Delta$. When a thread invokes `EXECUTE` on a vertex, it performs an aggregate fetch via `MULTI-FETCH` (as opposed to `FETCH`) to get all its Δ snapshots (lines 3 and 6), computes the new values for the Δ snapshots (lines 8-15), and then performs an aggregate store via `MULTI-STORE` to update the Δ snapshots (line 16). When one graph snapshot is fully processed (G_{stable}), it is replaced by another snapshot (G_{next}) so that the algorithm is always processing Δ snapshots in parallel (lines 28-35).

The above computation reordering is legal because processing of individual snapshots occur independently and hence, there are no dependences between iterations of the outer `for` loop. This means, upon unrolling the outer `for` loop, the inner `do-while` loops can be fused together by maintaining multiple versions of vertex values and capturing individual snapshot's convergence and vertex activations as described below.

4.3.2 Mutable Vertex Values

To incorporate FA, the in-memory representation of the evolving graph should support simultaneous processing of Δ graph snapshots. Hence, mutable vertex values are vector-expanded to store Δ values which are being computed simultaneously. If Δ snapshots are simultaneously processed in parallel, the memory consumption increases to $\mathrm{O}\big((\Delta \times |V|) + (p \times |E|)\big)$. This increase in memory consumption limits the degree up to which multiple graph snapshots can be processed together. As we will see in Section 4.6, we process 10 snapshots together for this reason.

Algorithm 6 Incorporating Fetch Amortization.

```
 1: function EXECUTE( GΔ)
 2:     for vid ∈ GET-ACTIVE-VERTEX(GΔ) do
 3:         vertices ← MULTI-FETCH(vid, GΔ)
 4:         nbrs ← ∅
 5:         for nid ∈
            GET-NEIGHBORS(vertices, GΔ) do
 6:             nbrs ← nbrs ∪
              MULTI-FETCH(nid, GΔ)
 7:         end for
 8:         for each vertex ∈ vertices do
 9:             old-value ← vertex.GET-VALUE( )
10:             comp-value ← f (vertex, nbrs)
11:             vertex.SET-VALUE(comp-value)
12:             if |old-value − comp-value| > ϵ
                then
13:                 ACTIVATE-NEIGHBORS(vertex)
14:             end if
15:         end for
16:         MULTI-STORE(vertices,GΔ)
17:     end for
18: end function

19: function MAIN(G, Δ)
20:     GΔ ← G[0 ... Δ − 1]
21:     INITIALIZE(GΔ)
22:     ACTIVATE-VERTICES(GΔ)
23:     do
24:         parallel-for all threads do
25:             EXECUTE(GΔ)
26:         end parallel-for
27:         barrier( )
28:         for each G_stable ∈ GΔ with
            no active vertex do
29:             OUTPUT-VERTEX-VALUES(G_stable)
30:             G_next ← GET-NEXT-GRAPH(G)
31:             INITIALIZE(G_next)
32:             ACTIVATE-VERTICES(G_next)
33:             GΔ ← GΔ \ {G_stable}
34:             GΔ ← GΔ ∪ {G_next}
35:         end for
36:     while there are
        unprocessed graphs in G
37: end function
```

4.3.3 Vertex Activations

During execution, not all Δ versions of the vertex may be activated to be processed since activations are dependent on computed values which may not be same for different versions. Hence, vertex activations for different versions need to be explicitly tracked which is done using additional bits in the worklist that maintains active vertices. While remote activation messages can be sent across machines as soon as they become available, multiple activations of same vertex for different versions can also be aggregated together. Hence, we send out the remote activation messages after all the Δ versions of the vertex are processed, i.e., at the end of the iteration which processes the vertex (line 15).

While synchronous processing model requires that only active vertices in a given iteration are processed in that iteration, asynchronous processing model relaxes this notion and tolerates computations on vertices even when they are not activate. In this case, activations can be tracked at vertex level which eliminates the need to maintain additional bits in the activation work-list.

4.3.4 Convergence Detection

To quickly determine whether or not processing has converged, individual machines either maintain a convergence flag which is set to false whenever vertices are activated to be processed in the next iteration, or rely on checking the activation queue. This convergence information is exchanged between machines, typically via parallel reduction, to determine whether all machines should stop processing. Since FA processes Δ snapshots in parallel, we maintain a vector of convergence flags, one for each snapshot being processed, and set a given flag to false whenever the corresponding version of any vertex is activated to be processed. Now, instead of exchanging the convergence information using scalar reduction, the vector-expanded convergence flags from all the machines are reduced together. Determining convergence separately for each of the Δ snapshots allows pipelined execution of snapshots where snapshots which are fully processed are replaced by new snapshots which are to be processed next (lines 28-35).

4.3.5 Caching & Message Aggregation

Distributed graph processing environments typically rely on caching to make remote values (vertices, etc.) locally available. When caches are used to store vertex values,

Benchmark	Overlap
PageRank	87.9 %
Single Source Shortest Path	99.3 %
Single Source Widest Path	80.2 %
Circuit Simulation	76.7 %
Heat Simulation	99.8 %
Graph Coloring	97.5 %

Table 4.6: Average % overlap of vertex values across consecutive graph snapshots of *Slashdot* input.

the overall fetches from remote machines is reduced. However, this benefit does not come for free – the cache management protocol itself generates additional messages to maintain the data values coherent. A major part of the additional messages are *invalidates* with *piggy-backed updates* (as in the case of ASPIRE & GraphLab). Hence, once the stores of multiple snapshots of a vertex are aggregated by fetch amortization, the corresponding cache protocol messages for those versions of the same vertex, going to the same destination, are also aggregated. This reduces the overheads incurred by the caching protocol itself. While fetch amortization can be directly incorporated in ASPIRE using the iterative execution algorithm presented in Algorithm 6, we have also integrated fetch amortization in GraphLab, details of which will be discussed in Section 4.5.

4.4 Processing Amortization

In our SSSP example from Section 4.1.2, comparing the final results for vertices of snapshots G_1 (Table 4.1) and G_2 (Table 4.2) reveals that values of vertices b, c, d, and e are the same for both snapshots. Vertex a is the only common vertex whose result value is not the same for the two snapshots; note vertices f and g are present only in G_2. Similarly,

on comparing the results computed for G_2 (Table 4.2) with G_3 (Table 4.3) we observe that resulting values for vertices b, c, d, e, f, and g are the same.

To further confirm the above observation, we measured the average percentage of vertex values that are found to be the same across consecutive snapshots for seven iterative algorithms (listed in Table 4.9) on the *Slashdot* graph. The results in Table 4.6 show that this is a high percentage – on an average across consecutive snapshots, over 76% of vertices had values for a given snapshot identical to those for the previous snapshot. Hence, we explore the following question:

Can we leverage the results (potentially partially computed) for previous graph snapshots to accelerate processing of later snapshots?

Moreover, when we compare *unstable values* i.e., the intermediate results, from step 2 in G_2 (Table 4.2) with final results of G_3 (Table 4.3), we observe that the values of most available vertices in G_1 (all except vertex a) are exactly same. This means, unstable results from previous snapshots may also be used in order to accelerate processing of later snapshots. This allows amortizing processing costs simultaneously while leveraging the first opportunity, i.e., processing consecutive snapshots in batches by using unstable results from previous snapshots to process later snapshots.

4.4.1 Processing Amortization via Feeding

Based upon the above observations we propose *processing amortization* technique in which the computed values (from stable and unstable vertices) for a given snapshot can

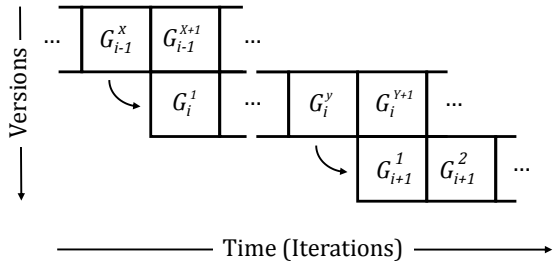


Figure 4.4: Effect of Processing Amortization.

be used to accelerate processing of the next snapshot by avoiding repeated computation of same values. In presence of fetch amortization, when processing of a snapshot finishes, the new snapshot which is included in the working set of snapshots being processed can be fed from its previous snapshot (analogous to pipelined processing), which might not have converged to its stable solution (see Figure 4.4). The advantage of doing this is twofold: it removes redundant computations so that the needed (non-redundant) processing can be performed sooner; and vertices stabilize faster because the computed values used from previous snapshot are already close to stability.

Processing Amortization (PA). While snapshot G_{i-1} is being processed, when processing of G_i is initiated for simultaneous processing, processing amortization *feeds* current vertex values from snapshot G_{i-1} into G_i as initializations to accelerate the processing of G_i.

Algorithm 7 shows the iterative processing with processing amortization. When processing of snapshot G_{stable} completely finishes (i.e., there are no active vertices), and the processing of a new snapshot G_{next} is initiated, instead of initializing vertices of G_{next} to their standard initial values, they are copied from the latest snapshot G_{prev} which is still being processed

(lines 14-15). This step involves copying the entire working set which includes the following three components: the vertex values mapped to the machines; activations for vertices; and, any caches holding vertex values. Vertices which are active for G_{prev} potentially have unstable values and hence, it is necessary to activate them for G_{next}.

Note that the values that are fed from G_{i-1} to G_i may not have stabilized yet as G_{i-1} is in the midst of processing (see Figure 4.4). However, as our experiments show, this optimization yields benefits because G_{i-1} has often made enough progress towards stability that it helps accelerate G_i's termination. On the other hand, in absence of fetch amortization, the values fed by PA from G_{i-1} to G_i are always stable.

Algorithm 7 Incorporating Process Amortization.

```
 1: function MAIN(G, Δ)
 2:     GΔ ← G[0 ... Δ − 1]
 3:     INITIALIZE(GΔ)
 4:     ACTIVATE-VERTICES(GΔ)
 5:     do
 6:         parallel-for all threads do
 7:             EXECUTE(GΔ)
 8:         end parallel-for
 9:         barrier( )
10:         for each G_stable ∈ GΔ with
            no active vertex do
11:             OUTPUT-VERTEX-VALUES(G_stable)
12:             G_prev ← GET-LATEST-
        GRAPH(GΔ)
13:             G_next ← GET-NEXT-GRAPH(G)
14:             INITIALIZE(G_next ← G_prev)
15:             ACTIVATE-VERTICES(G_next ←
        G_prev)
16:             GΔ ← GΔ \ {G_stable}
17:             GΔ ← GΔ ∪ {G_next}
18:         end for
19:     while there are
        unprocessed graphs in G
20: end function
```

4.4.2 Applicability & Correctness

Even though processing amortization shows potential to accelerate evolving graph processing, it is important to ensure that the technique guarantees correct results for each of the graph snapshots. To better understand the set of feasible algorithms, we identify

the characteristics of graph algorithms and then reason about correctness of each of the algorithms considered in our evaluation.

Mutation Properties

Feeding of values across consecutive snapshots can be viewed as mutation in the graph structure while the overall computation progresses. Hence, we study the impact of PA on correctness of results by reasoning about the behavior of graph algorithms when the structure of the graph is mutated, i.e., when vertices and edges are added and deleted to progress from one snapshot to the next. Determining properties in this manner is similar to the study performed in [13] related to incremental data flow analysis algorithms.

Note that vertex addition occurs when an edge with new end vertex is added. Similarly, vertex deletion occurs when all its edges get deleted. Hence, we model vertex addition/deletion via addition/deletion of its edges.

(A) Global Mutation Property: We define two properties that directly characterize the results obtained by the iterative algorithms based on their behavior and the values using which they start computing the results.

[P-INIT] Convergence to correct results is independent of vertex initializations.
[P-MONO] Intermediate results exhibit monotonic trend under a given ordering.

For algorithms exhibiting [P-INIT], when the graph structure mutates in the middle of computation, subsequent processing operates on vertex values coming from computations performed prior to the mutation. These values can be considered as new vertex initializations using which the processing converges to stable values. This means, feeding values

does not affect the correctness of results even when graph structure changes because the fed values are considered as vertex initializations which do not impact correct convergence; only the path to achieve final convergence changes by feeding different values. Hence, it is easy to see that [P-INIT] is a sufficient condition for an algorithm to be able to use PA correctly. However, note that [P-INIT] is not a necessary condition.

Algorithms exhibiting [P-MONO] can be reasoned about in a similar manner as monotonic data-flow functions [13] where the algorithms aim to achieve a maximum or a minimum fixed point based on a cost-metric. Such algorithms need to be carefully evaluated to determine whether processing can recover from local maxima/minima and correctly converge to global maxima/minima when edges are added and deleted in the middle of processing. We illustrate such a reasoning with the Connected Components algorithm which is based on iterative label propagation [152] as shown below.

$$v.value = \begin{cases} v.id & \text{... when } iteration = 0 \\ \min(v.value, \min\limits_{e \in \text{edges}(v)}(e.other.value)) & \text{... otherwise} \end{cases}$$

The above function starts with an approximation where every vertex is in a separate component and then iteratively merges components by picking the minimum component value available across the vertex's neighborhood. When edge addition occurs, the subsequent iteration computes over the vertices of the edge which incorporate the new neighbor's component value. If the two vertices have different component values, [P-MONO] ensures that the vertex with the larger value will get reassigned the smaller value from the new neighbor, effects of which are propagated throughout the graph. When edge deletion occurs,

the deleted edge can split a component into two separate components as shown in Figure 4.5a. In this case, recomputing the component value for vertex 2 incorrectly results in value 0 because its remaining neighbors, i.e., vertices 3 and 4, still exhibit the old component values. This means Connected Components can safely leverage PA only on growing graphs, i.e., those that evolve purely using addition of new edges.

(B) Local Mutation Property: Contrary to the above properties, we define [P-EDGE] at the level of vertex functions.

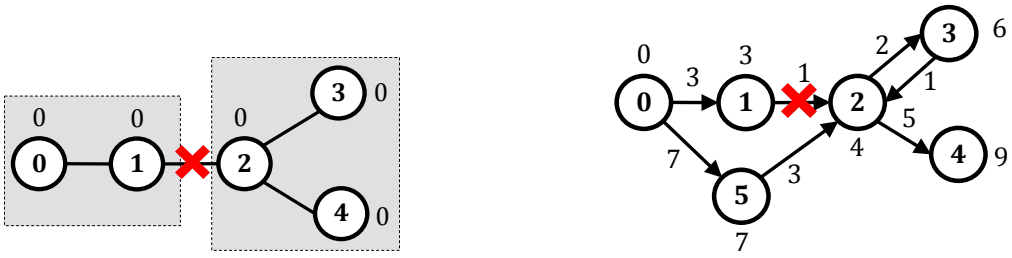
[P-EDGE] Computation for a vertex only depends on values coming from its edges.

Algorithms exhibiting [P-EDGE] majorly compute values based on graph structure and hence, they rely lesser on previously computed values. This allows the computation to self-correct values and propagate the corrections throughout the graph in iterative manner. We illustrate the effectiveness of [P-EDGE] using Single Source Shortest Paths (SSSP) algorithm; below are two variants for expressing SSSP which correctly compute shortest paths on a given graph snapshot.

$$\text{A) } v.value = \begin{cases} 0 & \text{... when } v \text{ is source} \\ \infty & \text{... when } iteration = 0 \\ \min(v.value, \min\limits_{e \in \text{inEdges}(v)}(e.weight + e.source.value)) & \text{... otherwise} \end{cases}$$

$$\text{B) } v.value = \begin{cases} 0 & \text{... when } v \text{ is source} \\ \infty & \text{... when } iteration = 0 \\ \min\limits_{e \in \text{inEdges}(v)}(e.weight + e.source.value) & \text{... otherwise} \end{cases}$$

The above functions initially set source to 0 and start with an approximation where all vertices are unreachable from the source, i.e., the path length is ∞. Then, they iteratively compute shorter paths for every vertex based on available shortest paths in its neighborhood [1]. The only difference between two variants is that while computing the new path value, variant A considers its previous path value whereas variant B does not, i.e., variant A does not exhibit [P-EDGE] whereas variant B does.



(a) Connected Components. (b) Single Source Shortest Paths.

Figure 4.5: Edge deletion examples.

When edge addition occurs, the target of newly added edges compute new values and if the newly added edge results in shorter paths, both the variants compute the corrected path, effects of which are iteratively propagated throughout the graph. When edge deletion occurs, however, the two variants behave differently as illustrated in Figure 4.5b. Upon deletion of edge (1, 2), variant A still incorrectly retains its shortest path as 4 (note that simply resetting its value to ∞ does not resolve the issue because of the back-edge from vertex 3). Variant B, on the other hand, corrects its path value to 10 ($0 \rightarrow 5 \rightarrow 2$) due to [P-EDGE]. This means, variant A can safely leverage PA only on growing graphs whereas variant B can do so on evolving graphs which include deletion of edges too.

[1] Edge weights are positive for path problems.

Algorithm	Vertex Function
PR	$v.rank \leftarrow 0.15 + 0.85 \times \sum_{e \in \text{inEdges}(v)} e.source.rank$
SSSP	$v.path \leftarrow \min_{e \in \text{inEdges}(v)} (e.source.path + e.weight)$
GC	$conflict \leftarrow \bigvee_{e \in \text{edges}(v)} ((v.color = e.other.color) \text{ and } (v.id < e.other.id))$ $decrease \leftarrow$ true if $\exists c < v.color$ s.t. $\forall_{e \in \text{edges}(v)} (e.other.color \neq c)$ if $conflict = true$ or $decrease = true$ then: $v.color \leftarrow c$: where $\forall_{e \in \text{edges}(v)} (e.other.color \neq c)$
HS	$v.value \leftarrow \frac{\sum_{e \in \text{inEdges}(v)} C_1 \times e.source.value + C_2}{\lvert \text{inEdges}(v) \rvert}$
SSWP	$v.path \leftarrow \max_{e \in \text{inEdges}(v)} (\min(e.source.path, e.weight))$
CS	$v.value \leftarrow \frac{\sum_{e \in \text{inEdges}(v)} e.weight \times e.source.value}{\sum_{e \in \text{inEdges}(v)} e.weight}$

Table 4.7: Various vertex-centric graph algorithms.

It is interesting to note that [P-EDGE] is neither a necessary nor a sufficient property to guarantee correctness of results when using PA. However, it is important to reason about [P-EDGE] because it defines a subset of vertex algorithms for which PA may be safely used.

Correctness of Algorithms

Using the properties described above, we now study each of our benchmark algorithms considered in our evaluation (listed in Table 4.9) to discuss about their correctness while incorporating PA. Table 5.3 shows the vertex functions for each of the algorithms (vertex initializations are eliminated for simplicity).

(A) Shortest & Widest Paths:

Our Single Source Shortest Paths (SSSP) algorithm is the variant B from the above discussion that exhibits [P-EDGE]. As discussed above, it can safely leverage PA. Single Source

Widest Paths (SSWP), similar to SSSP, exhibits [P-EDGE]. However as discussed above, [P-EDGE] is not a sufficient condition to guarantee correctness while using PA and hence, we will see that edge deletions can impact the correctness of results of SSWP. When edge addition occurs, the target vertex recomputes its value based on the available edges and the newly added edge can increase the fed path value (due to monotonic nature of `max`), which is further observed by its neighbors to be propagated throughout the graph. Hence, correct results are guaranteed when PA is used with edge additions. However, when edge deletion occurs, the target vertex can incorrectly compute its path value to be same as the fed value from an alternate path which previously passed through itself, disallowing the vertex to step out of its previous approximation. Note that this does not occur in SSSP because positive edge weights ensure that such incorrect cyclic feeding does not occur. Hence, we evaluate PA on SSWP with growing evolving graphs (i.e., allowing edge additions only).

(B) Graph Coloring:

Graph Coloring (GC) is an interesting algorithm which effectively corrects vertex color values based on the neighbors values. In particular, when edge addition occurs, if the new neighboring vertices have the same color values, the vertex with the lower id updates its value to a new color value based on its neighbors, effects of which iteratively propagate throughout the graph. When edge deletion occurs, the neighboring vertices are reset to default color values (to guarantee minimality) and recomputed to receive new color values based on the remaining neighborhood. Hence, even after structural mutations are observed across snapshots, the fed incorrect values are treated as initializations using which correct coloring solution is computed, i.e., GC exhibits [P-INIT] allowing PA to be safely used.

(C) PageRank, Heat Simulation, Circuit Simulation:

It is well known that for algorithms like PageRank (PR), the initial vertex values do not matter [37], i.e., they exhibit [P-INIT]. This allows PR to safely leverage PA. Similarly, both Circuit Simulation (CS) and Heat Simulation (HS) do not require vertex initializations except for their source/ground vertices which are not computed upon during processing; hence, they also exhibit [P-INIT] and can safely incorporate PA while processing.

Beyond the set of benchmarks considered in our evaluation, we studied many different vertex programs in order to ensure applicability of PA. Various algorithms including, but not limited to, K-Means, Wave Simulation, Reachability, Maximal Independent Set, Triangle Counting, NumPaths, BFS, and Diameter can safely incorporate PA, ensuring its wider applicability.

4.5 Graph Processing Systems

The amortization techniques proposed in this work are independent of any specific graph processing system (and its internals) that can process a given graph snapshot at a time. Various graph processing systems have been developed across different processing environments, some of which include Pregel [86], GraphLab [85] [44], ASPIRE [129], GraphX [45] and GraphChi [73]. To evaluate the efficacy of our proposed techniques, we incorporate our amortization techniques in two of those systems, namely GraphLab and ASPIRE. We briefly discuss these two frameworks and how we incorporate the amortization techniques to process evolving graphs [2].

[2]Details about system runtime, caching, programming, performance numbers, etc. for each of the frameworks can be found in their respective publications.

4.5.1 ASPIRE

ASPIRE [129] uses an object based DSM where the objects (vertices and edges) are distributed across the cluster such that there is exactly one global copy of each object in the DSM. The machine-level caches are kept coherent using a directory based protocol which provides strict consistency [46]. Since the vertex values are usually small, invalidates are piggy-backed with the updated data. Other standard optimizations like *Work Collocation* (or locality in [30]) and *Message Aggregation* (similar to bulk transfer in [80]) are enabled.

The graph is partitioned across machines to minimize edge-cuts for reducing communication using the well-known ParMETIS [62] partitioner. The runtime holds a flag per vertex indicating whether a vertex is active for the next iteration. For a particular vertex, the programmer can determine whether to activate neighbors on out-going, in-coming, or both edges. After each iteration, a check is made to see whether computation for a graph snapshot is complete. If not, the activation vectors are exchanged between the machines so that each machine aggregates the vertices it needs to work on in the next iteration. Before starting the next iteration, a check is made to ensure that all the invalidates have been acted upon. This guarantees that updated values are visible and correctly used in the next iteration.

In order to incorporate FA, we take advantage of ASPIRE's single-writer model by having the same thread process multiple snapshots of the same vertex at the same time. This allows remote fetches, stores and all the cache protocol messages to be automatically aggregated across multiple snapshots of same vertices. To transfer objects to and from the DSM, the runtime provides `DSM-Fetch`, `DSM-MultiFetch` and `DSM-Store`, `DSM-MultiStore`

APIs. This hides the internal details of the runtime and allows parallel algorithms to directly execute over the DSM.

4.5.2 GraphLab

GraphLab [85] [44] is a popular distributed graph processing framework which provides shared memory abstractions to program over a distributed environment. It provides synchronous and asynchronous engines to process the graph and allows users to express computations in gather-apply-scatter (GAS) model. Similarly to ASPIRE, boundary vertices are cached (named as *ghost vertices*) on different machines to make them locally available and the graph is partitioned using a balanced p-way partitioning strategy to minimize cuts and balance computation.

Since processing is done on a set of machines, FA is used to tolerate network latencies especially for ghost vertices, which need to be continuously synchronized. This is done by vectorizing the vertex values and updating the vertex functions to process multiple versions of same vertex. PA is incorporated by feeding values from the most recent snapshot to the next snapshot; we only allow this feeding when all the executing snapshots become stable because GraphLab prevents dynamic changes to all vertex values and graph structure while the engine is processing.

4.5.3 Other Graph Processing Frameworks.

Incorporating both amortization techniques in various graph processing frameworks requires knowledge about their processing engines and how computations are expressed (vertex centric, edge centric, etc.). This is because different frameworks are de-

Evolving Graph	V	E
Twitter-25 [17]	8.4B	25.5B
Delicious-100 [31]	1.6B	15.2B
DBLP-100 [79]	92.7M	963.8M
Amazon-100 [81]	144M	299.8M
StackEx-100 [117]	27.6M	70.7M
Epinions-100 [88]	9.4M	47.5M
Slashdot-70 [43]	2.1M	7.5M

Table 4.8: Real-world evolving graphs taken from KONECT [70] repository for experiments.

Benchmark	Type
PageRank (PR) Single Source Shortest Path (SSSP) Single Source Widest Path (SSWP) Graph Coloring (GC)	Graph Analytics & Mining
Circuit Sim. (CS) Heat Sim. (HS)	PDE

Table 4.9: Benchmark programs used in experiments.

veloped in order to exploit different processing environments and hence, they operate in different manner. However, we identify two simple techniques which can be easily used to incorporate both transformations in a new processing framework.

For FA, the values associated with graph structure (vertices and edges) can be vectorized to hold multiple versions in order to allow simultaneous processing of multiple snapshots. Also, the user defined processing function for a given algorithm can be vectorized so that it operates over a vector of vertex/edge values instead of traditional scalar values. For PA, the simplest technique is to orchestrate pipelined batch processing of graph snapshots by invoking the processing engine multiple times, each time using values from previous results. However, in order to fully leverage PA, the processing engine itself needs to be modified so that it becomes aware of this pipelined processing technique and feeding of values occurs internally as soon as processing of any graph snapshot ends.

4.6 Experimental Evaluation

4.6.1 Experimental Setup

Real-world Datasets. Our experiments use real-world evolving graphs taken from the KONECT [70] repository. Table 4.8 lists the input data sets and their sizes that range from 25.5 billion to 7.5 million edges. Similar to as in [49], we develop snapshots by first dividing the entire evolution into half; the first snapshot is this midpoint and then subsequent snapshots are created by batching the remaining edge additions and deletions. The number of snapshots is included in each input's name in Table 4.8. We also create a synthetic graph called StackEx+D to simulate edge deletions using StackEx. In this case, the edges to be deleted are randomly selected from a given graph snapshot and the next snapshot is constructed using equal number of edge deletions and additions.

Benchmarks. To evaluate our techniques, we use a wide range of modern applications (as listed in Table 4.9) that belong to important domains such as scientific simulation and social network analysis. These applications are based on the vertex centric model where each vertex iteratively computes a value (e.g., colors for GC, ranks for PR [94], and shortest paths for SSSP) and the algorithm terminates when values become stable, i.e., all the algorithms were run fully until completion.

System. The experiments were run on a 16-node cluster running CentOS 6.3, Kernel v2.6.32 with each node having two 8-core Xeon E5-2680 processors (16 cores) operating at 2.7GHz and 8X4GB (32 GB) DDR3-1600MHz memory. The nodes are connected via 56Gbit/s FDR InfiniBand interconnect.

The first set of performance results (i.e., entire Figure 4.6) are averaged (arithmetic mean) across 10 runs and since the observed deviation was typically less than 2%, the remaining results are averaged across 3 runs.

4.6.2 Performance of FA & PA in GraphLab/ASPIRE

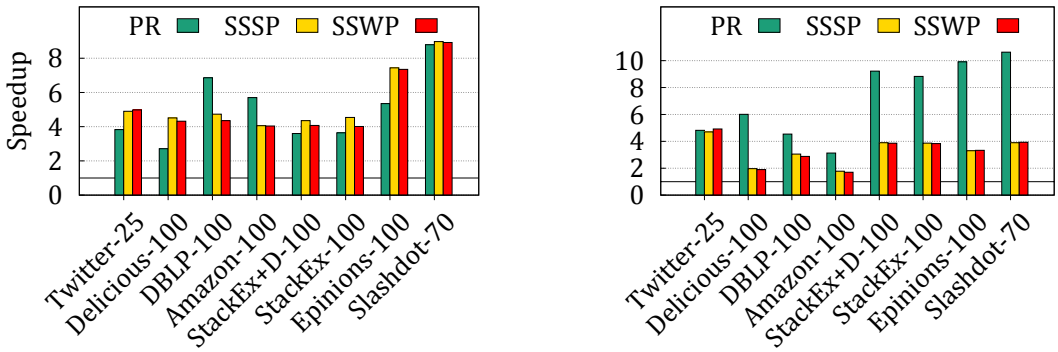
We evaluate the benefits of the proposed techniques in distributed systems, ASPIRE and GraphLab [3], by comparing the performance of the following versions of the benchmarks:

- **GraphLab/ASPIRE** is the original version that does not employ our amortization techniques.
- **FA** is the version in GraphLab/ASPIRE that employs fetch amortization.
- **PA** is the version in GraphLab/ASPIRE that employs processing amortization.
- **FA+PA** is the version in GraphLab/ASPIRE that employs both techniques.

We carried out this performance comparison using all six benchmarks in ASPIRE and three benchmarks in GraphLab – PR, SSSP, and SSWP. For FA and FA+PA, we choose the number of snapshots to be simultaneously processed (Δ) to be 10 (more experiments with varying Δ are further presented in Section 4.6.4.). Since ParMETIS requires high amount of memory to partition very large graphs, we ran ASPIRE with first 40/20 snapshots for DBLP/Delicious graphs and do not evaluate using the Twitter-25 graph. GraphLab's synchronous engine outperforms its asynchronous engine in all cases and hence, we report the times obtained from its synchronous engine.

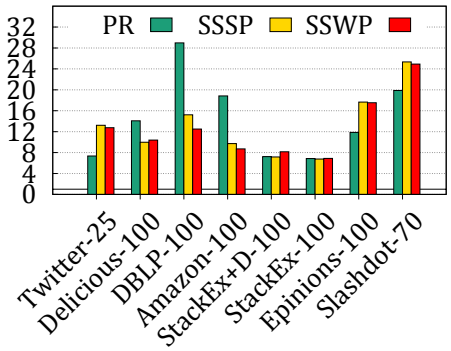
[3] We used the latest available version of GraphLab (v2.2).

Figure 4.6 and Figure 4.7 show the speedups achieved by FA, PA, and FA+PA when incorporated in GraphLab and ASPIRE respectively. On an average, FA in GraphLab achieves 5.2× speedup over original GraphLab and further addition of PA yields additional 2× speedup over FA in GraphLab. Similarly, on an average, adding FA to ASPIRE yields a speedup of 4.06× over the original version of ASPIRE and further addition of PA results in additional 7.86× speedup over FA alone.



(a) Speedups by FA.

(b) Speedups by PA.

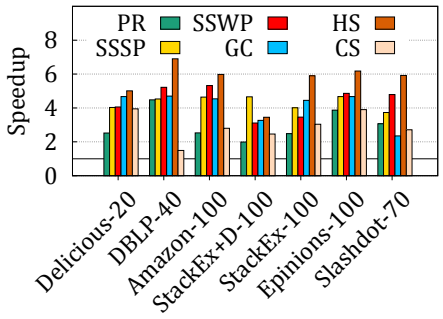


(c) Speedups by FA+PA.

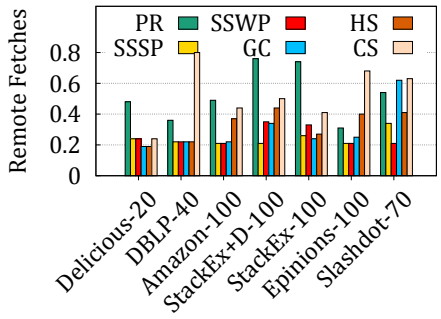
Figure 4.6: Speedups achieved by FA, PA, and FA+PA in GraphLab. The execution times (in sec) for PR/SSSP on original version of GraphLab (Baseline) are: 4,117/2,900 for Twitter-25 and 7,148/1,490 for Delicious-100.

Figure 4.7(b)(d)(f) show the reduction in remote fetches and vertex computations performed when amortizations are enabled. Overall, the speedups achieved by the amortization techniques vary across the benchmark-input combinations. However, it is interesting to note that the improvement from FA is usually higher for SSSP and SSWP than for PR even though the remote fetches saved by FA are more for PR than both SSSP and SSWP. On an average, FA in ASPIRE gives 4.3/4.4× speedups over the original versions for SSSP/SSWP whereas for PR the corresponding speedup is 3×. This is because PR is more compute-intensive compared to SSSP and SSWP; hence, even though reductions in remote fetches are higher for PR, vertices take more time to stabilize to their final values for PR compared to SSSP and SSWP, leading to more work to be done in each intermediate iteration. Similarly, speedups for HS are lower with PA compared to those with FA. This is because HS is more sensitive to structural changes in the graph and hence, the fed values are less useful, causing fair amount of work to be performed (as can be seen in Figure 4.7(d)) to reach stability.

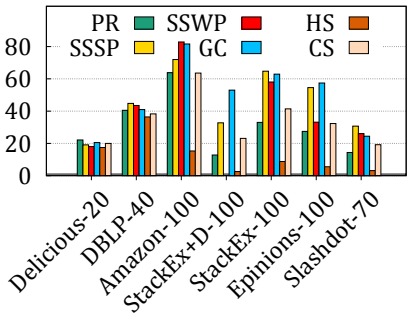
It is interesting to observe that in GraphLab, FA+PA performs better than PA whereas the trend is reverse in ASPIRE on large graphs. This is due to the fundamental difference in how the remote vertex values are locally maintained in these two systems. In FA+PA, when a new snapshot gets included to be processed with the other snapshots, vertex values for the new snapshot get fed from the previous snapshot. In ASPIRE, this change in value is reflected on remote machines in the subsequent iteration (immediately after the inclusion of the new snapshot) when the values are accessed, which causes the computation to wait for the changed values to become available. For PA, on the other



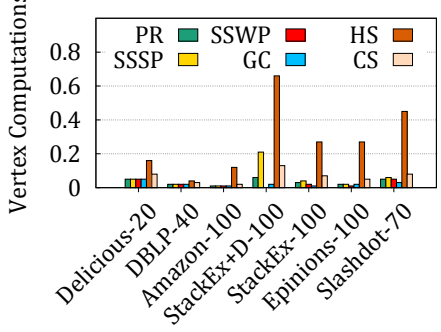
(a) Speedups by FA.



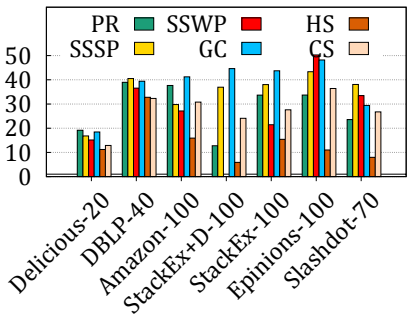
(b) Remote fetches by FA.



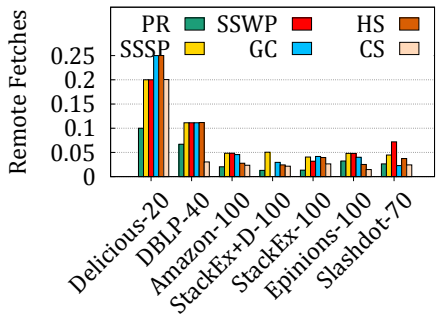
(c) Speedups by PA.



(d) Vertex Computations by PA.



(e) Speedups by FA+PA.



(f) Remote fetches by FA+PA.

Figure 4.7: Speedups achieved by FA, PA, and FA+PA in ASPIRE (a,c,e); Reduction in Remote Fetches for FA, Vertex Computations for PA, and Reduction in Remote Fetches for FA+PA over FA (b,d,f). The execution times (in sec) for PR/SSSP on original version of ASPIRE (Baseline) are: 16,245/7,244 for Delicious-20 and 9,282/2,340 for DBLP-40.

hand, the fed values are the same and hence, remote machines already have correct values in their software caches when processing of the new snapshot begins. The above behavior does not occur in GraphLab because remote vertex values are kept consistent with local copies as soon as new edges get added since they represent computation dependences.

We achieve consistent speedups across all inputs, including both with and without edge deletions. On StackEx+D-100 (which includes edge deletions), on an average, FA in GraphLab and ASPIRE gives 3.6× and 3.2× speedups respectively over their original versions and further addition of PA achieves additional speedups of 1.8× and 7.9× respectively. No data is provided for PA and FA+PA for SSWP on StackEx+D-100 because PA is not applicable on SSWP in the presence of edge removals.

While Figure 4.6 reports the overall speedups, we also separated the processing times from the evolving graph management times. When only the processing times are considered, on an average, FA speeds up processing by 5.3× in GraphLab while PA achieves additional speedups of 2.6×.

Note that simultaneously processing too few graph snapshots fails to fully realize the benefits of FA while processing excessive numbers of graph snapshots can also lead to suboptimal performance. Simultaneously processing too many graph snapshots causes many vertices to be active even though only some of the snapshots of those vertices are active. This delays processing for a given snapshot because different snapshots of other vertices need to be processed. Therefore the number of graph snapshots that are simultaneously processed is a parameter that must be tuned for a given distributed environment. As discussed in Section 4.3.1, the degree up to which multiple graph snapshots can be processed together

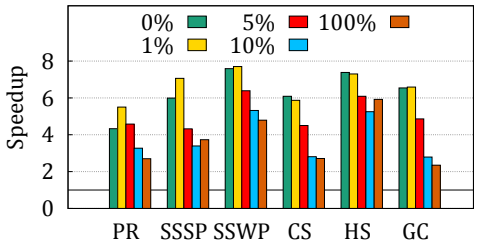
is bound by available memory. Based upon our experiments by varying the number of snapshots that are simultaneously processed we found that simultaneously processing 10 snapshots gives good performance overall. Therefore the experimental data reported above used these settings when running FA and FA+PA.

From the results we observe that amortization optimizations are very effective as they significantly improve the performance of both GraphLab and ASPIRE. The results for FA clearly show the need to tolerate communication latencies by amortizing fetches in both GraphLab and ASPIRE; the results for PA clearly show that amount of computation is significantly reduced by feeding values across snapshots.

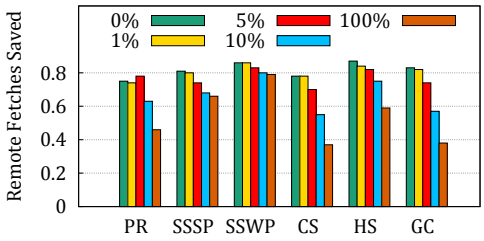
4.6.3 Sensitivity to Cache Size

Both caching and amortization optimizations reduce remote fetches. Hence, it is important to study their interplay and how the benefits achieved from the amortization techniques are affected when caching is available in the system. To study how the benefits of amortizations vary with caching, we experiment using ASPIRE and vary the machine-level software cache sizes to be 100%, 10%, 5%, 1% of the number of vertices as well as when no cache is used. The ASPIRE runtime incorporates machine-level caches to make the remote vertices locally available for processing. The cache coherence protocol provides strict consistency [46] and cache invalidates are piggy-backed with updated vertex values.

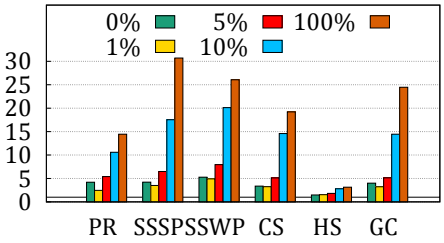
Since earlier we observed that the benefits achieved from FA and PA are similar across all inputs, we now select one of the inputs (Slashdot) to further perform this sensitivity study in detail. Finally, we ran all six applications in Table 4.9 for each configuration.



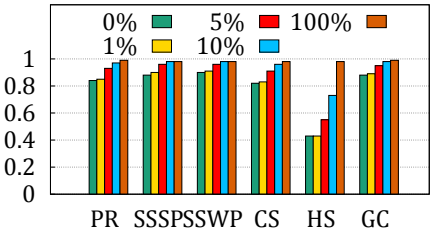
(a) Speedups by FA.



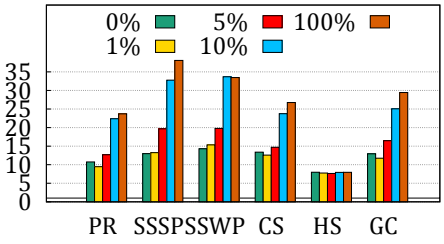
(b) Remote fetches saved by FA.



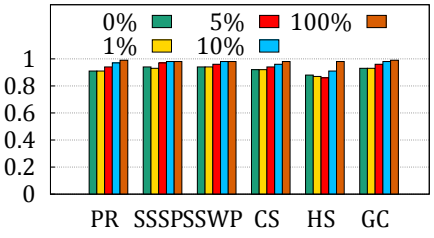
(c) Speedups by PA.



(d) Remote fetches saved by PA.



(e) Speedups by FA+PA.



(f) Remote fetches saved by FA+PA.

Figure 4.8: Effect of varying cache size on FA, PA and FA+PA (a,c,e); Remote fetches saved by FA,PA and FA+PA (b,d,f)

From Figure 4.8a, we first observe that, although both caching and FA can reduce remote fetches, they complement each other quite well. Consider the case where the cache size is 100%. When FA uses 10 snapshots plus caching, it provides speedups ranging from 2.35× to 5.92× over the ASPIRE Baseline [4] with caching enabled. For the 10%, 5% and 1% cache size, the corresponding speedups range from 2.79× to 5.32×, 4.32× to 6.39× and 5.5× to 7.7×. In other words, FA provides substantial speedups *beyond caching* by reducing remote fetches. The main reason for these speedups is the significant number of remote fetches saved by FA (shown in Figure 4.8b); as cache size decreases from 100% to 0%, the average savings increase from 54% to 82%. It is interesting to note that the speedups with FA are higher with 1% compared to other cases which is due to the inefficiency of the small cache which provides higher miss-rate; on an average, cache with size 1% provides 84.56% miss-rate compared to 34.78% to 10.12% miss-rate provided by caches with size 5% to 100%. Hence, the overheads of maintaining smaller cache of 1% size effectively enables larger scope for savings which are captured by FA.

Alternatively, we also find that the 10% cache size is enough to capture most of the benefits of caching — further increasing the cache size from 10% to 100% would reduce the average miss rates only from 14.82% to 10.12% for the Baseline algorithm. The corresponding average speedups of FA plus caching over Baseline plus caching increase from 3.7× to 3.8× as the cache size decreases from 100% to 10%; moreover, this 10× increase in cache size does not yield much additional benefits. Note that the 5% cache size, on the other hand, is not enough to capture benefits comparable to that with the 10% cache size.

[4]Baseline refers to the ASPIRE version without FA and PA.

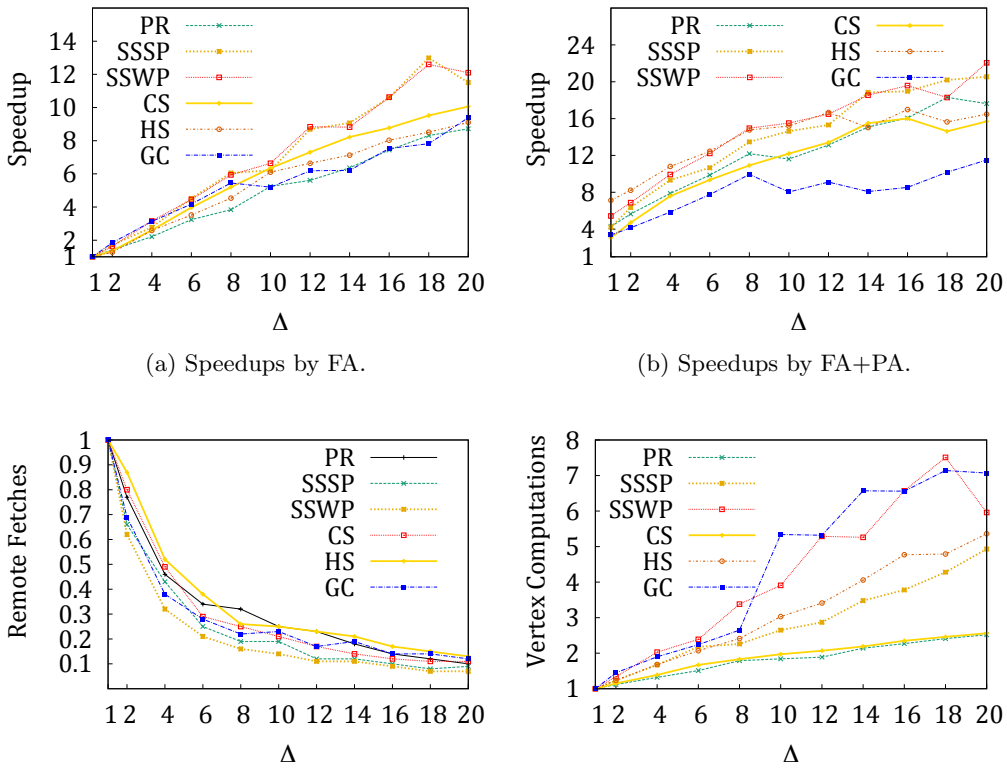
The average hit rate for the Baseline algorithm with the 5% cache size is 34.78%, and hence, the average speedups of FA plus caching over Baseline plus caching increase to 5.12×.

Figure 4.8c shows the speedups achieved by PA with varying cache sizes. The speedups increase with increase in cache sizes; this is mainly because of the elimination of redundant computations by PA which makes the computation more communication bound, allowing larger caches to provide more speedups by saving more remote fetches (shown in Figure 4.8d). Again, the overheads of smaller cache with size 1% along with higher miss-rate reduces the speedups achieved compared to other cases.

Finally, as seen in Figure 4.8e FA+PA provides highest speedups with different cache sizes mainly because of the very high amount of remote fetches saved (as shown in Figure 4.8f) by both the amortization strategies.

4.6.4 Sensitivity to number of snapshots (Δ)

We also varied the number of snapshots processed simultaneously, i.e., Δ from one to twenty. As discussed in Section 4.3.1, the degree up to which multiple graph snapshots can be processed together is bound by available memory. Hence, we do not go beyond simultaneously processing 20 snapshots. We again select Slashdot and use all six benchmarks to perform this sensitivity study. In the first set of experiments, we disabled the machine-level caches to decouple the effects of caching from our amortization techniques. Later, we enabled the machine-level caches and vary both, the number of snapshots and the cache sizes, to study the interplay between them.



(a) Speedups by FA.

(b) Speedups by FA+PA.

(c) Remote fetches by FA.

(d) Vertex Computations by FA+PA.

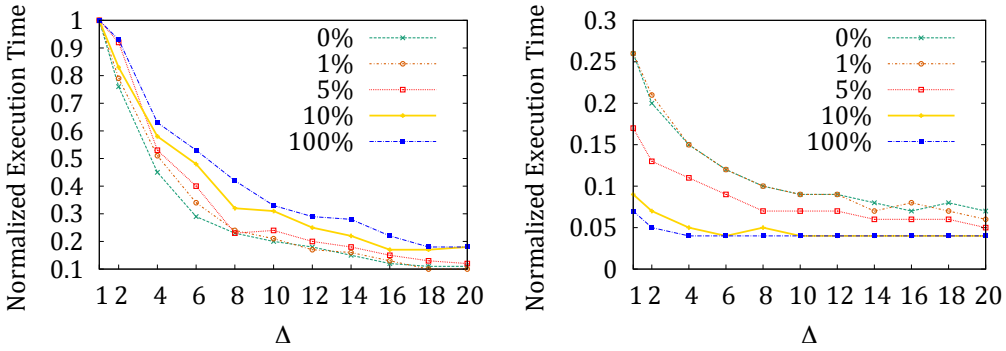
Figure 4.9: Effect of varying Δ on FA (a) and FA+PA (b); Reduction in Remote Fetches for FA (c); and Increase in Vertex Computation for FA+PA (d).

The effects of varying Δ on the overall performance are shown in Figure 4.9. Clearly, our approach scales well with the number of snapshots. Figure 4.9a shows that as Δ increases, the benefits of FA show up. The reduction in remote fetches achieved by FA in those cases is shown in Figure 4.9c; as we can see, the speedups are mainly achieved due to this reduction.

It is interesting to note that bulk of the reduction in remote fetches is achievable when $\Delta = 8$. Hence, we observe nearly linear speedup for FA up till $\Delta = 8$. Beyond $\Delta = 8$, some benchmarks exhibit saw-tooth trend where the benefits of increasing Δ by small steps

are not clearly visible. The reason for this is the decline in reduction of remote fetches for corresponding intermediate Δ steps beyond $\Delta = 8$ which can be seen in Figure 4.9c.

Figure 4.9b shows the benefits achieved by FA+PA with increase in Δ. As we already know, PA eliminates redundant computations and FA with increased Δ reduces the overall remote fetches performed. It is interesting to note from Figure 4.9d that as Δ increases, the amount of processing required increases compared to PA with $\Delta = 1$.



(a) FA execution times for different cache sizes. (b) FA+PA execution times for different cache sizes.

Figure 4.10: Effect of varying Δ for different cache sizes on FA (a) and FA+PA (b).

This is mainly because for $\Delta > 1$, unstable values are fed across snapshots, and these unstable values are further away from the final values for new snapshots, hence requiring more work to stabilize these newly fed snapshots. However, this increase in processing is overshadowed by savings in remote fetches, hence resulting in an overall increase in speedups with increase in Δ. Note that the speedups for GC nearly flatten-out for higher Δ values because of the high impact of structural changes across snapshots which are far apart in evolution resulting in more color changes across the graph.

Finally, Figure 4.10a and Figure 4.10b show the execution times for PR for varying Δ with different cache sizes on FA and FA+PA respectively, normalized w.r.t. $\Delta = 1$ without PA. As we can see, the performance benefits are achieved in all cases with different Δ and cache size combinations. As expected, increasing Δ reduces the overall execution time; again, majority of performance benefits are achieved when Δ is increased up to a certain point, after which, lesser additional benefits are seen – in particular, while the performance continues to improve beyond Δ = 8-10, the improvement is not significant. This means, the difference in execution times for higher values of Δ is very less and hence, we can achieve majority of the performance benefits from our techniques without determining the ideal Δ value for each of the cache sizes.

4.6.5 Sensitivity to similarity in snapshots

While PA is useful to eliminate redundant computations by feeding values across snapshots, we study its effectiveness when consecutive graph snapshots are made structurally dissimilar. In this experiment, we vary the percentage of overlapping edges, i.e., edges that are same, across consecutive snapshots from 90% (high overlap) to 50% (low overlap) and measure the speedups achieved by PA with and without FA ($\Delta = 10$). We again select Slashdot and use all six benchmarks to perform this sensitivity study. Since we are interested in effectiveness of the amortization strategy itself, we enabled machine-level caches to hold the required remote vertices on each machine.

The effects of varying the percentage of overlapping edges are shown in Figure 4.11. As we can see, PA accelerates the overall processing in all cases; even if consecutive snapshots

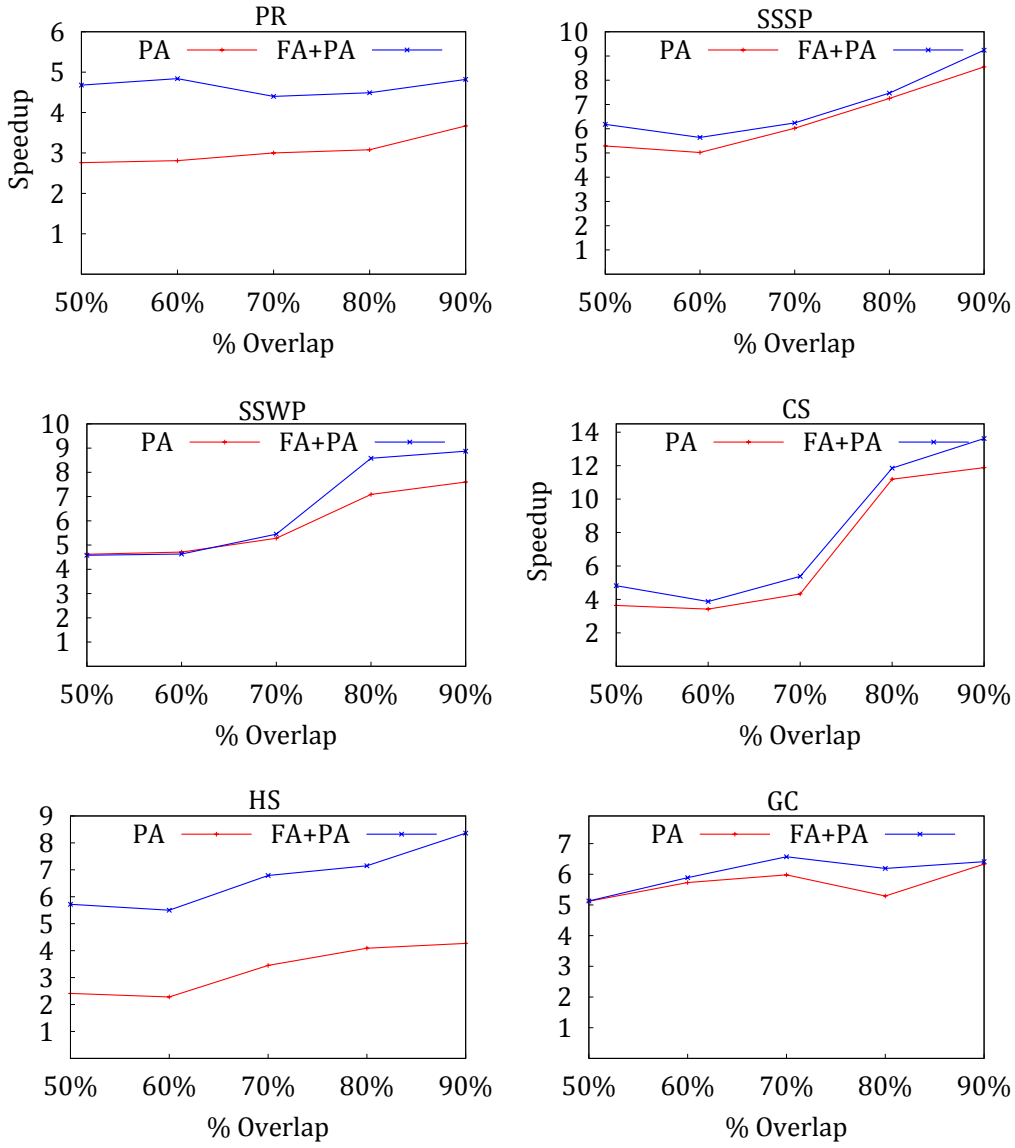


Figure 4.11: Effect of varying the overlap across consecutive snapshots on PA and FA+PA.

are structurally dissimilar by up to 50%, feeding of values provides a better initialization by eliminating redundant computations and hence, convergence is achieved faster. As expected, the speedups increase with increase in structural overlap mainly because the fed values are more close to final results and hence, require lesser computation. The speedups do not increase as clearly for PR and GC as those compared to SSSP, SSWP, CS and HS when

the overlap increases from 50% to 90%. This is because the number of remote fetches performed by PR and GC without the amortization techniques is high enough to minimize the difference between relative reduction achieved using the amortization techniques as similarity varies.

4.6.6 Comparison with Chronos

Table 4.10 highlights the differences between our amortization techniques and Chronos [49]. While Chronos incorporates batch scheduling and incremental processing, it is designed to improve the L1 data cache and last level cache performance in a shared memory environment. As shown in [49], the benefits are lower in a distributed setting (an average speedup of 4.06×) mainly because it is constrained by the network overheads. Our amortization techniques on the other hand collectively reduce the network overheads and provide average speedups of 28.9× and 10.3× in ASPIRE and GraphLab.

Aspect	**Chronos**	**Amortization Techniques**
Temporal Layout	Enhance locality for cache performance	Expose structural similarity to aggregate remote fetches
Snapshots	Static	Dynamic
Batch Processing	Fixed batching	Variable batching using pipelining
Feeding Unstable Values	Absent	Present
Message Aggregation	Absent	Present

Table 4.10: Amortization Techniques v/s Chronos.

Since Chronos itself is a temporal graph processing system, we directly compare the performance of PA with the *incremental processing* (IP) technique. In presence of FA, IP is performed by feeding stable values from the most recent snapshot in the previous batch to all the snapshots in the next batch. This requires Chronos to wait for all the

snapshots in the previous batch to finish before the next batch can begin execution. PA, on the other hand, enables faster processing by feeding unstable (partially computed) values which are available from the most recent snapshot. This allows the batch of snapshots being processed to be dynamically adjusted as snapshots finish processing, eliminating the need to stall processing of future snapshots.

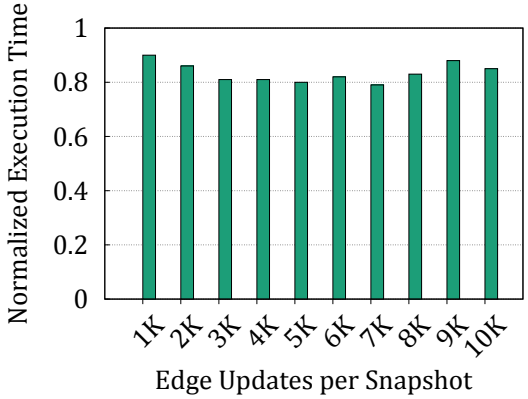


Figure 4.12: Execution times for PR using FA+PA normalized w.r.t. FA+IP.

We incorporated IP in ASPIRE by processing snapshots batch after batch so that only final results from the most recent snapshots in the previous batch are fed to all the snapshots in the next batch. Figure 4.12 shows the execution times for PR using FA+PA normalized w.r.t. that using FA+IP across multiple evolving graphs created from StackEx input by varying the number of edge updates required to determine new snapshots. On an average, FA+PA performs 16.5% faster than FA+IP mainly because PA does not stall processing of snapshots which allows those snapshots to be computed in parallel as other snapshots are being processed.

4.7 Summary

In this chapter, we leveraged the asynchrony to accelerate evolving graph processing using two optimizations. Fetch Amortization reduces remote fetches by aggregating similar messages that get exposed due to computation reordering. Processing Amortization accelerates termination of iterative algorithms by carefully using incremental computations. Our experiments with multiple real evolving graphs and graph algorithms on a 16-node cluster demonstrate that, on average fetch amortization speeds up the execution of GraphLab and ASPIRE by 5.2× and 4.1× respectively. Amortizing the processing cost yields additional average speedups of 2× and 7.9× respectively. In the next chapter, we will consider another case of dynamic graphs, called *streaming graphs*, and develop techniques based on the incremental computation strategy to process them efficiently.

Chapter 5

Streaming Graph Processing

In the previous chapter we saw how evolving graphs can be efficiently processed using computation reordering and incremental computation techniques. In this chapter, we will focus on another scenario involving dynamic graph processing, namely *streaming graph processing*. To provide timely responses to online data analytic queries, streaming graph processing incorporates continuous computation over dynamic graphs as their structure keeps on changing. This means, there are no distinct graph snapshots to be processed as in evolving graphs; however, a single graph whose structure keeps on changing rapidly needs to be processed efficiently to compute results for the most updated graph structure.

In this chapter, we will exploit the algorithmic asynchrony to develop an incremental processing technique so that computations can benefit from previously calculated results. However, naïvely using incremental computation can impact: a) the correctness of the solution; and, b) the performance of the overall processing. Hence, we will develop a *dynamic dependence* based incremental processing technique that efficiently computes query results while simultaneously providing 100% correctness guarantees.

The need to analyze streaming graphs has led to the development of systems such as Tornado [112] and others. The core idea of these systems is to interleave iterative processing with the application of batches of updates to the graph. The iterative processing maintains an *intermediate approximate result* (intermediate for short) of the computation on the *most recent* version of the graph. When a query arrives, the accurate result for the *current version* of the graph where all batched updates have been applied is obtained by performing iterative computation starting at the intermediate results. In other words, computations at the vertices with edge updates are performed directly on their most recent intermediate values computed before the updates arrive. This style of processing leverages *incremental computation* to achieve efficiency. The intuition behind it is straightforward: the values right before the updates are a better (closer) approximation of the actual results than the initial vertex values and, hence, it is quicker to reach convergence if the computation starts from the approximate values.

Problems

However, the above intuition has an implicit assumption that is often overlooked: an intermediate value of a vertex is indeed closer to the actual result than the initial value *even when the graph mutates*. We observe that this assumption always holds for strictly growing graphs if the graph algorithm performs a *monotonic computation* (e.g., SSSP, BFS, Clique, label propagation algorithms, etc.), because adding new edges preserves the existing graph structure on which intermediate values were computed. However, if graph is mutated via edge deletions, the graph structure changes may *break monotonicity* and invalidate the intermediate values being maintained.

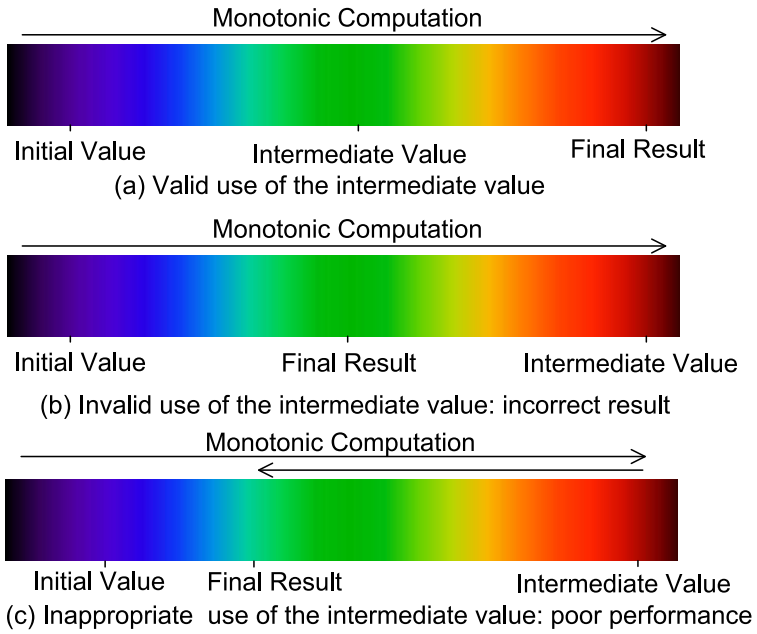


Figure 5.1: Three different scenarios w.r.t. the use of intermediate values after an edge update.

Figure 5.1 depicts three scenarios *w.r.t.* the use of approximate results in the processing of streaming graphs. Since we focus on monotonic graph algorithms, each spectrum shows the unidirectional change of vertex values. Let us examine how the computation is impacted by the use of intermediate values. In Figure 5.1(a), the intermediate result is between the initial value and the final accurate result. This scenario is a valid use of intermediate result because it is closer to the final result than the initial value. Performing monotonic computations on a strictly growing graph falls in this category.

Figure 5.1(b), however, shows an opposite scenario where the final result is between the initial and the intermediate values. An edge deletion may fall into this category. To illustrate this situation, consider a path discovery algorithm where the intermediate path computed before the deletion (*i.e.*, intermediate result) no longer exists and the new path to be discovered (*i.e.*, final result) is "worse" than the intermediate path. If the algorithm

only updates the vertex value (*i.e.*, path discovered) when a new "better" path is found, the algorithm will stabilize at the non-existent old path and converge to an *incorrect* result.

Figure 5.1(c) shows a slightly different scenario than Figure 5.1(b) where the algorithm, despite being monotonic, is also self-healing. In this case, the computation goes "backward" after an edge deletion and finally stabilizes at the correct result. However, starting the computation at the intermediate result is clearly *unprofitable*. It would have taken much less effort to reach the correct result had the computation was restarted at the initial value. Detailed examples illustrating these cases are presented in Section 5.1.

It may appear that the problem can be solved by always resetting the value of a vertex to its initial value at the moment one of its incoming edges is deleted and making its computation start from scratch. This approach would still lead to incorrect results because computations at many other vertices are *transitively* dependent upon the deleted edge; thus, only resetting the value at the deleted edge does not handle these other vertices appropriately (Section 5.1). Resetting *all* vertex values solves the problem at the cost of completely disabling incremental computation and its benefits.

Our Approach

In this chapter, we present a novel runtime technique called KickStarter that computes a *safe* and *profitable* approximation (*i.e.*, trimmed approximation) for a small set of vertices upon an edge deletion. KickStarter is the *first technique* that can achieve safety and profitability for a general class of monotonic graph algorithms, which compute vertex values by performing *selections* (discussed shortly). After an edge deletion, computation starting at the trimmed approximation (1) produces correct results and (2) converges at least at the same speed as that starting at the initial value.

The key idea behind KickStarter is to *identify* values that are (directly or transitively) impacted by edge deletions and *adjust* those values before they are fed to the subsequent computation. A straightforward way to do so is to *tag* the target vertices of the deleted edges and to carefully propagate the tags to the rest of graph. The values for all the tagged vertices are reset (to the initial value) to ensure correctness.

Although tagging guarantees correctness, it is performed conservatively as it is unaware of how the intermediate results are dynamically computed. Hence, it typically tags vertices excessively, leaving only a small set of vertices with usable approximate values. To overcome this drawback, KickStarter characterizes the *dependences* among values being computed and tracks them as the computation progresses. However, tracking dependences online can be very expensive; how to perform it efficiently is a significant challenge.

We overcome this challenge by making an observation on monotonic algorithms. In many of these algorithms, the value of a vertex is often *selected* from one single incoming edge, that is, the vertex's update function is essentially a *selection function* that compares values from all of the incoming edges (using max, min, or other types of comparisons) and selects one of them as the computed value of the vertex. This observation applies to all monotonic algorithms that we are aware of, including the eight algorithms listed later in Table 6.1. This feature indicates that the current value of a vertex only *depends on* the value of *one single* in-neighbor, resulting in dependences that can be efficiently tracked.

Upon an edge deletion, the above dependence information will be used first to find a small set of vertices impacted by the deleted edges. It will also be used to compute safe approximate values for these vertices. The detailed explanation of the dependence tracking

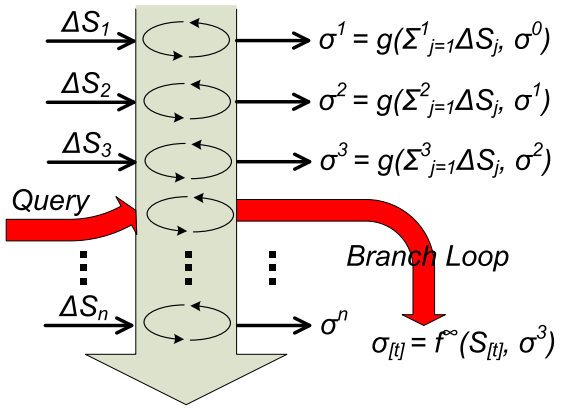


Figure 5.2: Streaming graph processing.

and the trimming process can be found in Section 5.2. We have evaluated KickStarter using four monotonic algorithms and five large real-world graphs. Our results show that KickStarter not only produces correct results, but also accelerates existing processing algorithms such as Tornado [112] by 8.5–23.7×.

The rest of the chapter is organized as follows. Section 5.1 discusses the correctness and performance issues involved in streaming graph processing. Section 5.2 describes the incremental processing techniques based on trimming via dynamic dependence tracking. Finally Section 5.3 presents the experimental setup and detailed analysis of results.

5.1 Background and Motivation

In a typical streaming iterative graph processing system such as Tornado [112], the implementation employs a main loop that continuously and iteratively processes the changing graph to compute intermediate results for the most recent snapshot of the graph. Figure 5.2 illustrates this processing loop. While graph update requests constantly flow in, updates are batched (ΔS_i) and not applied until the end of an iteration. Upon a user

```
1: function SSWP(Vertex v)
2:     maxPath ← 0
3:     for e ∈ INEDGES(v) do
4:         p ← MIN(e.src.path,
5:                 e.weight)
6:         if p > maxPath then
7:             maxPath ← p
8:         end if
9:     end for
10:    v.path ← maxPath
11: end function
```

(a) Single source widest path.

```
1: function SSSP(Vertex v)
2:     minPath ← ∞
3:     for e ∈ INEDGES(v) do
4:         p ← e.src.path+
5:                 e.weight
6:         if p < minPath then
7:             minPath ← p
8:         end if
9:     end for
10:    v.path ← minPath
11: end function
```

(b) Single source shortest path.

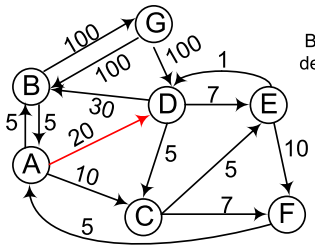
Figure 5.3: Two path discovery algorithms.

query (for a certain property of the graph), the main loop forks a *branch loop* that uses the intermediate values computed at the end of the previous completed iteration of the main loop (*e.g.*, σ^3 in Figure 5.2) as its starting values. The branch loop then iteratively processes the graph until convergence. The final results are returned to the user.

5.1.1 Problem 1: Incorrectness

We use a Single Source Widest Path (SSWP) example to show that naïvely using the above algorithm in presence of *edge deletions* can lead to incorrect results. SSWP solves the problem of finding a path between the designated source vertex and every other vertex in a weighted graph, maximizing the weight of the minimum-weight edge in the path. It has many applications in network routing where the weight of an edge represents the bandwidth of a connection between two routers. The algorithm can be used to find an end-to-end path between two Internet nodes that has the maximum possible bandwidth.

Figure 5.3(a) illustrates a vertex-centric implementation of SSWP. Next, we show that, for the simple graph given in Figure 5.4(a), feeding the computation with either the



(a) A simple graph.

Before deletion →

A	B	C	D	E	F	G
∞	20	10	20	7	7	20
A→D deleted						
∞	20	10	20	7	7	20
Correct Result						
∞	5	10	5	5	7	5

(b) Computation using intermediate values after the deletion of A→D; shown in blue are the correct result.

A	B	C	D	E	F	G
∞	20	10	20	7	7	20
A→D deleted						
∞	20	10	0	7	7	20
∞	20	10	0	5	7	20
∞	20	10	20	5	7	20
∞	20	10	20	7	7	20

(c) Computation using the initial value for D; values changed are in red.

Figure 5.4: Using either the intermediate or the initial value for vertex D leads to incorrect results (which are underlined); the initial value for each vertex is 0.

intermediate value or the initial value in presence of deletion of edge $A \rightarrow D$ generates incorrect results. Figure 5.4(b) reports the value of each vertex before and after deletion when the approximate value is used (*i.e.*, 20) for vertex D. Before the edge update, $A \rightarrow D$ is the key edge that contributes to the value 20 at D and G. After it is deleted, G and E become the only in-neighbors of D. Since G's value is still 20, D's value is not updated; so are the values of the other vertices. The computation stabilizes at the pre-deletion values, generating incorrect results.

Figure 5.4(c) shows that resetting the value of D to its initial value 0 does not solve the problem either. Clearly, despite the change, D's value will be incorrectly updated back due to the influence from G. The reason behind these problems is that the three vertices B, D, and G form a cycle and the computation of their values depends on each other. Only setting D's value is not enough to correct the wrong influence from the other nodes.

Precisely, for a given widest path $u \rightarrow v$, the vertex function maintains the invariant that $v.path \leq u.path$. Hence, for a cycle $v \rightarrow w \rightarrow ... \rightarrow k \rightarrow v$ in the graph, the maintained invariant is $k.path \leq ... \leq w.path \leq v.path$. Suppose the actual solution for this entire path is $u.path = v.path = w.path = k.path = m$. When the edge $u \rightarrow v$ is deleted,

the vertex function computes a new value for v using its remaining incoming edges, one of which is $k \to v$. At this point, v would still receive value m from edge $k \to v$. If m is greater than the values coming from v's other in-neighbors, $v.path$ will still be set to m. This is incorrect since the value m of vertex k was originally computed from the value of v itself.

Similar incorrect behaviors can be observed for ConnectedComponents (CC) (see Table 5.3) — if there is a cycle, all vertices in the cycle can end up having the same component ID which would create wrong influence after edge deletions.

Motivation from Real Graphs

Figure 5.5 shows the numbers of vertices that have wrong values in the query results for SSWP and CC on the LiveJournal and UKDomain graphs (see Table 5.2 for details of the graphs). Edge updates are batched in our experiments. At the end of each batch, we send a query that asks for the values of all vertices in the new version of the graph obtained by applying the current batch of edge updates. The details of the experiment setup can be found in Section 5.3.

The vertices that have wrong results are identified by using the results of KickStarter as an *oracle*. Observe that the number of such vertices is noticeably high. Furthermore, the inaccuracies for each batch are carried over into the main processing loop, affecting future query results – this can be seen from the fact that there are increasing numbers of vertices with wrong values as more queries (batches) are performed (processed).

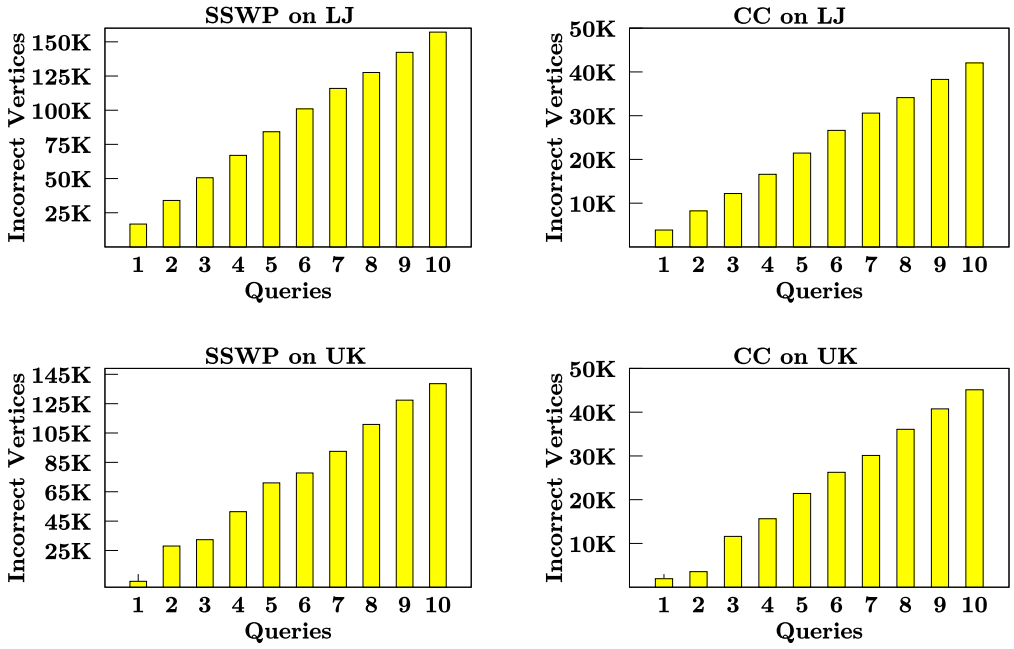
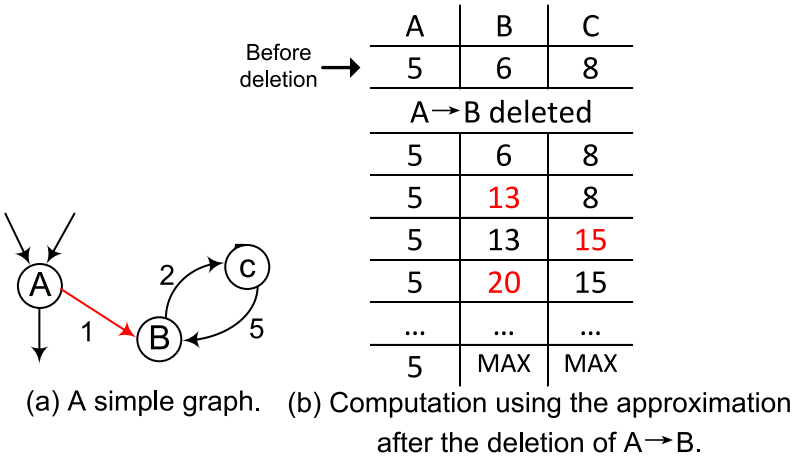


Figure 5.5: Numbers of vertices with incorrect results.

5.1.2 Problem 2: Degraded Performance

Consider the SSSP algorithm in Figure 5.3(b). While this algorithm produces the correct result, it would have severe performance problem if the approximate value is used upon the deletion of edge $A \rightarrow B$ in the graph shown in Figure 5.6(a). The deletion of the edge renders the vertices B and C disconnected from the rest of the graph. Using the intermediate values 6 and 8 for the forward computation would bump up these values at each iteration (Figure 5.6(b)); the process would take a large number of iterations to reach the final (correct) result (`MAX`). This is exactly an example of the scenario in Figure 5.1(c).



A	B	C
5	6	8
A→B deleted		
5	6	8
5	13	8
5	13	15
5	20	15
...	...	...
5	MAX	MAX

Figure 5.6: While using the intermediate value for vertex B yields the correct result, the computation can be very slow; the initial value at each vertex is a large number `MAX`.

5.1.3 How to Distinguish Algorithms

The above examples showed two types of monotonic algorithms, those that may produce incorrect results and others that produce correct results but may have significantly degraded performance in the presence of edge deletions. While the problems in both types are caused by cycles in the graph, different algorithm implementations may lead to different consequences (*i.e.*, incorrectness vs. performance degradation). We observe that the key difference between them is in their vertex update functions. In the first type of algorithms such as SSWP, the update function only performs *value selection* (*i.e.*, no computation on the value is done). Hence, when a cycle is processed, there is a potential for a value being propagated along the cycle without being modified and eventually coming back and inappropriately influencing the vertex that loses an edge, producing incorrect results.

The update function of the second type of algorithms performs computation on the selected value. For example, SSSP first selects a value from an in-neighbor and then adds the edge weight to the value. The addition ensures that when a value is propagated

along the cycle, it appears as a different value at the vertex via which the original value had entered the cycle. In this case, the vertex function disallows cyclic propagation and hence, upon deletion, it becomes impossible for the algorithm to stabilize at a wrong value. To summarize, whether the update function of an algorithm contains actual computation can be used as a general guideline to reason about whether the algorithm can produce incorrect values or would only cause performance problems.

5.1.4 Correcting Approximations using KickStarter

For both the correctness and performance issues, KickStarter trims the approximation such that correct results can be computed efficiently. For our SSWP example, KickStarter generates A(∞) B(5) C(10) D(0) E(5) F(7) G(5) using which, it would take the computation only one iteration to converge at the correct results. Similarly, for our SSSP example, KickStarter generates A(5) B(`MAX`) C(`MAX`) which is exactly the correct result. The detailed trimming algorithm will be presented in Section 5.2.

5.2 Trimming Approximations

This section describes KickStarter's trimming techniques. We begin with an overview and then discuss the algorithm.

5.2.1 KickStarter Overview

Given a graph $G = (V, E)$, let an approximation $A_G = a_0, a_1, ..., a_{n-1}$ be a set of values for all vertices in V, *i.e.*, $\forall a_i \in A_G,\ a_i$ is the value for vertex $v_i \in V$ and $|A| = |V|$.

For an iterative streaming algorithm S, a *recoverable approximation* A_G^S is an approximation with which the final correct solution on G can be computed by S. A simple example of a recoverable approximation is the set of *initial* vertex values, *i.e.*, values at the beginning of the processing. Note that due to the *asynchronous* nature of streaming algorithms, at any point during the computation, multiple recoverable approximations may exist, each corresponding to a distinct execution path leading to convergence with same correct result.

KickStarter Workflow

For monotonic streaming algorithms (e.g., SSWP), the approximation maintained in the presence of edge additions is *always recoverable*. Hence, upon a query, running the computation on the updated graph with the approximation before the edge addition always generates the correct result. However, when edge deletion occurs, the approximation before a deletion point may be irrecoverable. Thus, to generate a recoverable approximation, we add a *trimming* phase right after the execution is forked (i.e., the branch loop in Figure 5.2) for answering a user query. In this phase, the current approximation from the main loop is trimmed by identifying and adjusting *unsafe* vertex values. The trimmed approximation is then fed to the forked execution.

After the query is answered, the result from the branch loop is fed back to the main loop as a new approximation to accelerate the answering of subsequent queries.

Technique Overview

KickStarter supports two methods for trimming. The first method identifies the set of vertices s possibly affected by an edge deletion using a tagging mechanism that also exploits

algorithmic insights. For the vertices in s, their approximate values are *trimmed off* and reset to the initial values (*e.g.*, a large value for SSSP and 0 for SSWP). This method guarantees *safety* by conservatively tagging values that *may* have been affected. However, conservative trimming makes the resulting approximation less *profitable*.

In the second method, KickStarter tracks dynamic dependences among vertices (*i.e.*, the value of which vertex contributes to the computation of the value of a given vertex) *online* as the computation occurs. While tracking incurs runtime overhead, it leads to the identification of a much smaller and thus a more precise set of affected vertices s. Furthermore, because the majority of vertices are unaffected by edge deletions and their approximate values are still valid, trimming uses these values to compute a set of safe and profitable approximate values that are closer to the final values for the vertices in s.

Our presentation proceeds in following steps: Section 5.2.2 presents the first approach where trimming is performed by tag propagation; Section 5.2.3 presents the second approach where dynamic value dependences are captured and trimming is performed by calculating new approximate values for the affected vertices. An argument of safety and profitability is provided in Section 5.2.5.

5.2.2 Trimming via Tagging + Resetting

A simple way to identify the set of impacted vertices is vertex tagging. This can be easily done as follows: upon a deletion, the target vertex of the deleted edge is tagged using a set bit. This tag can be iteratively propagated — when an edge is processed, KickStarter tags the target of the edge if its source is tagged. The value of each tagged vertex is set back to its initial value in the branch loop execution.

While tagging captures the transitive impact of the deleted edge, it may tag many more vertices than is necessary. Their values all need to be reset (*i.e.*, the computation done at these vertices is not reused at all), although the approximate values for many of them may still be valid. To illustrate, consider the example in Figure 5.4. Upon the deletion of edge $A \rightarrow D$, this approach will tag all the vertices in the graph, even though approximate values for at least C and F are valid.

To further reduce the number of tagged vertices, KickStarter relies on *algorithmic insights* in propagating the tag across the vertices. The intuition here is to tag a vertex only if any of its in-neighbors that actually *contributes to* its current value is tagged. Since determining where the contribution comes from requires understanding of the algorithm itself, the developer can expose this information by providing a vertex-centric propagation function. For example, the following function shows tagging of vertices in SSWP:

$$tag(v) \leftarrow \bigvee_{\substack{e \in inEdges(v) s.t. \\ min(e.weight, e.source.value) = v.value}} tag(e.source) \tag{5.1}$$

Our insight is that in a typical monotonic algorithm, the value of a vertex is often computed from a single incoming edge. That is, only one incoming edge offers contributions to the vertex value. For example, in SSSP, the value of a vertex depends only on the smallest edge value. This observation holds for all monotonic graph algorithms that we are aware of, including the seven listed in Table 6.1. For these algorithms, the computation is essentially a *selection* function that computes the vertex value by selecting single edge value.

Algorithms	Selection Func.
Reachability	*or()*
ShortestPath, ConnectedComponents, MinimalSpanningTree, BFS, FacilityLocation	*min()*
WidestPath	*max()*

Table 5.1: Monotonic algorithms & their aggregation functions.

At the moment tagging is performed, an edge deletion has already occurred and all its impacted values have already been computed. Here we want to understand which edge contributes to the value of a vertex. The propagation function essentially encodes a "dynamic test" that checks "backward" whether applying the selection on a particular edge can lead to the value. However, since it is a backward process (*e.g.*, that guesses the input from the output), there might be multiple edges that pass this test. To guarantee safety, if the source of any of these edges is tagged, the vertex needs to be tagged. This is reflected by the $\vee$ (or) operator in Eq. 5.1.

Use of this technique in our example no longer tags vertices A and C. However, while propagation functions reduce the number of tagged vertices, tagging is still a "passive" technique that is not performed until a deletion occurs. This passive nature of tagging dictates that we must reset values for all the tagged vertices although many of them have valid approximate values at the time of tagging — KickStarter cannot determine safe approximate values for these vertices during tagging. For example, in Figure 5.4, even though vertex F has the correct approximate value 7, it gets tagged and then its value has to be reset to 0. In fact, F receives the same value from two different paths: $A \rightarrow C \rightarrow F$ and $A \rightarrow D \rightarrow E \rightarrow F$. Tagging does not distinguish these paths and hence, it ends up marking F regardless of the path the tag comes from.

Next, we discuss an alternative mechanism that *actively* captures the "which neighbor contributes to a vertex value" relationships, making it possible for us to compute *safe approximate values* for the impacted vertices.

5.2.3 Trimming via Active Value Dependence Tracking

In this approach we employ *active*, "always-on" dependence tracking, regardless when and where edge deletions occur. Through the recorded dependences, we can precisely identify the vertices impacted by a deleted edge. More importantly, the captured dependences form a data slice, following which safe approximate values can be computed. In contrast, tagging was not active but rather turned on only when a deletion occurred and thus it cannot *compute* approximate values.

Value Dependence

We first formalize a *contributes-to* relation ($\mapsto$) to capture the essence of this type of value dependences. Given two vertices u and v, $u \mapsto v$ if there is an edge from u to v and u's value individually contributes to the value of v. A transitive closure $\mapsto^*$ over $\mapsto$ thus involves all transitive contributes-to relationships. Based on $\mapsto^*$, we formalize a *leads-to* relation ($\xrightarrow{LT}$) to capture the desirable transitive dependences as described above. For two vertices u and v, $u \xrightarrow{LT} v$ iff. (1) $u \mapsto^* v$ and (2) $v \not\mapsto^* u$.

The second condition is important because it ensures that computation of v is not based on vertices whose values were computed using v's previous values (*i.e.*, a dependence cycle). Our goal is to guarantee safety: if a safe approximate value needs to be calculated for v upon an edge deletion, the calculation must avoid the effects of v's previous values

by not considering any incoming neighbor u such that $v \mapsto^* u$. It is actually a property of a monotonic graph algorithm — to guarantee convergence, the algorithm maintains monotonicity invariants across values of each vertex and values of the neighbors. For example, computation in SSSP has the invariant that if $u \mapsto v$, then v's value is no smaller than u's value. All the algorithms listed in Table 6.1 maintain such monotonicity invariants.

Note that the *leads-to* relation is a subset of the set of normal *flow* data dependences. A flow dependence, induced by a write followed by a read operating at the same location, may or may not give rise to a *leads-to* relationship, since $\xrightarrow{LT}$ is defined at a high level over vertices of the graph.

While $\xrightarrow{LT}$ may be automatically computed by runtime techniques such as dynamic slicing [149, 147], such techniques are often prohibitively expensive, incurring runtime overhead that causes a program to run hundreds of times slower. Since graph algorithms are often simple in code logic, we leverage the developer's input to compute $\xrightarrow{LT}$. We design a simple API `contributesTo` that allows the developer to expose *contribute-to* ($\mapsto$) relationships when writing the algorithm. For example, for SSWP, the edge on which the vertex value depends is the last edge that triggers the execution of Line 7 in Figure 5.3(a). The developer can add an API call right after that line to inform KickStarter of this new relationship which led to computation of the new value.

Note that we only rely on the developer to specify direct $\mapsto$ relationships, which incurs negligible manual effort. The transitive closure computation is done automatically by KickStarter. The freedom from dependence cycles is actually provided by monotonicity.

Representing `Leads-to` Relation as Dependence Trees As individual *contributes-to* relationships are profiled, these relationships form a dependence graph $D = (V^D, E^D)$, which shares the same set of vertices as the original graph $G = (V, E)$. Each edge in D represents a $\mapsto$ relationship and the edge set E^D is a subset of E. We maintain one single dependence graph across iterations. However, dependence profiling is not accumulative — if a new value of a vertex is computed and it results from a different in-neighbor, a new dependence edge is added to *replace* the old dependence edge for the vertex.

This dependence graph D encodes the $\xrightarrow{LT}$ relation and has two crucial properties. (1) It is *acyclic* – this follows the definition of *leads-to* relation which ensures that if there is a directed path from u to v in D, then there must be no directed path from v back to u in D. (2) Every vertex $v \in V^D$ has at most one incoming edge, *i.e.*, if $u \mapsto v$, then $\forall w \in V^D \setminus \{u\}, w \not\mapsto v$. This can be derived from the fact that the selection function only selects one edge to compute the vertex value and the computation of a new value at the vertex replaces the old dependence edge with a new edge.

The above properties imply that D is a set of *dependence trees*. Figure 5.7(a) shows a dependence tree for the SSWP algorithm in Figure 5.4 before deletion of $A \to D$.

Computing New Approximate Values

Taking the set of dependence trees as input, KickStarter computes new approximate values for the vertices that are affected by deletions. First of all, KickStarter identifies the set of vertices impacted by a deleted edge. This can be done simply by finding the subtree rooted at the target vertex of the deleted edge.

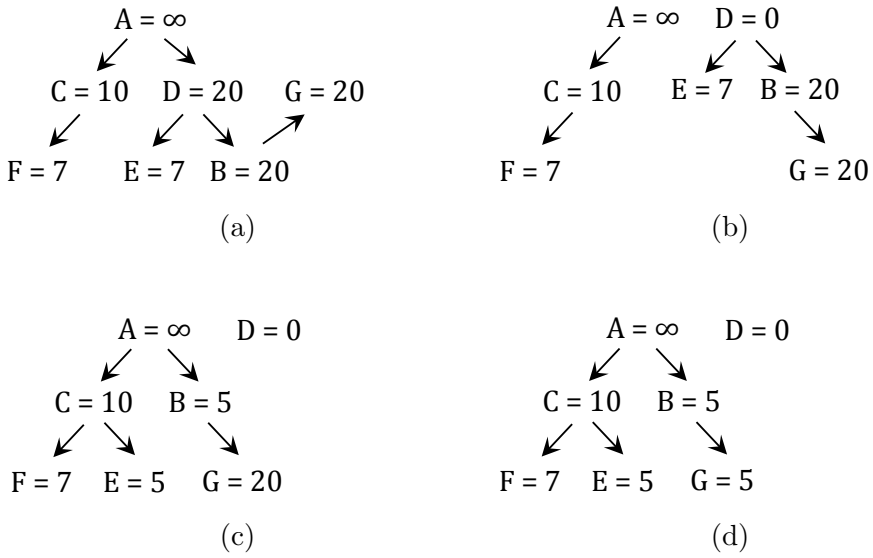


Figure 5.7: (a) Dependence tree for Figure 5.4(a) before the edge deletion; (b)-(d) trimming reorganizes the dependence tree.

To compute values for profitability, KickStarter employs three strategies: (1) it ignores deletions of edges which do not have corresponding dependence edges in D. This is safe because such edges did not contribute to the values of their target vertices; (2) if a deleted edge does have a corresponding dependence edge in D, KickStarter computes a *safe alternate value* for its target vertex. While resetting the approximate value of the vertex is also safe, KickStarter tries to compute a better approximate value to maximize profitability; and (3) once a safe alternate value is found for the vertex, KickStarter may or may not continue trimming its immediate children in D, depending on whether this new approximate value disrupts monotonicity.

Since the first strategy is a straightforward approach, we focus our discussion here on the second and third strategies.

Finding Safe Approximate Values Given a vertex affected by a deletion, KickStarter finds an alternate approximate value that is safe. Our key idea is to *re-execute* the update function on the vertex to compute a new value, starting from the target vertex of the deleted edge. One problem here is that, as we have already seen in Figure 5.4, there may be cycles (*e.g.*, B, G, and D in Figure 5.4(a)) in the actual graph (*not* the dependence graph) and, hence, certain in-neighbors of the vertex may have their values computed from its own value. This cyclic effect would make the re-execution still produce wrong values.

To eliminate the affect of cycles, KickStarter re-executes the update function at vertex v on a subset of its incoming edges whose source vertices do not depend on v. More precisely, we pass a set of edges e such that $v \not\mapsto^* e.src$ into the update function to recompute v's value. In Figure 5.4, after $A \rightarrow D$ is deleted, we first re-execute the SSWP function in Figure 5.3(a) at vertex D. The function does not take any edge as input — neither $G \rightarrow D$ nor $E \rightarrow D$ is considered since both the values of G and E depend on D in the dependence tree shown in Figure 5.7(a). D's value is then reset to 0.

Determining this subset of incoming edges for vertex v can be done by performing a dependence tree traversal starting at v and eliminating v's incoming edges (on the original graph) whose sources are reachable. However, such a traversal can be expensive when the sub dependence tree rooted at v is large. Hence, we develop an inexpensive algorithm that conservatively estimates reachability using the *level information* in the dependence trees.

For vertex v, let $level(v)$ be the level of v in a dependence tree. Each root vertex gets level 0. The tree structure dictates that $\forall w : v \mapsto^* w$, $level(w) > level(v)$. Hence, as a conservative estimation, we construct this subset of incoming edges by selecting every

edge such that the level of its source vertex u is $\leq$ the level of v. This approach guarantees safety because every in-neighbor w of v such that $v \mapsto^* w$ is excluded from the set. It is also lightweight, as the level information can be maintained on the fly as the tree is built.

When to Stop Trimming Once a safe value is found for a vertex, KickStarter checks whether the value can disrupt monotonicity. For example, if this value is higher (lower) than the previous vertex value in a monotonically decreasing (increasing) algorithm, the monotonicity of the current value is disrupted, which can potentially disrupt the monotonicity of its children vertices. In such a case, the resulting approximation may not be recoverable yet because cycles in the original graph can cause the effects of the previous value to inappropriately impact the computation at the children vertices. Hence, trimming also needs to be done for the children vertices.

On the other hand, if the monotonicity for the current vertex is not disrupted by the new value, the trimming process can be safely terminated because the approximate value for the current vertex is recoverable. Since the current vertex is the only one that contributes to the values of its children vertices, the values of the child vertices would become recoverable during the forward graph computation. As an example, for SSWP, if the old value for a vertex v is v_{old} and its new value is v_{new}, whether to continue the trimming process can be determined by the following rule:

$$continue \leftarrow \begin{cases} true & \text{... if } v_{new} < v_{old} \\ false & \text{... } otherwise \end{cases} \tag{5.2}$$

Example As new approximate values are computed, the dependence trees are adjusted to reflect the new dependences. Figure 5.7(b)-(d) show how the structure of the dependence tree in Figure 5.7(a) changes as trimming progresses. First, the subset of incoming edges selected for D, referred to as D^s, is an empty set, and hence, D's value is reset to 0 (*i.e.*, the initial value) and D becomes a separate root itself (Figure 5.7(b)) because it does not depend on any other vertex. Since this value is smaller than its old value (20), monotonicity is disrupted and thus D's immediate children, E and B, need to be trimmed. E^s consists of the incoming edges from C and D. The re-execution of the update function gives E a safe value 5, making E a child of C in the dependence tree (Figure 5.7(c)). Similarly, B^s consists of the incoming edges from A and D. B then receives the safe value 5, making itself a child of A in the tree (Figure 5.7(c)). Similarly, trimming continues to G, which receives a safe approximate value 5 from B (Figure 5.7(d)).

Putting It All Together: The Parallel Trimming Algorithm Trimming can be done in parallel on vertices since the computations involved in determining safe approximate values are confined to a vertex and its immediate neighbors. Hence, trimming itself can be expressed as vertex-centric computation and performed on the same graph engine.

Algorithm 8 presents the overall vertex-centric trimming algorithm. It first creates a subset of incoming edges (Lines 2-9) which can be used to generate a safe approximate value. Then it executes the same vertex function used to perform the actual graph computation to find a safe approximate value (Line 12). Note that when this is done, the value dependence exposed by the developer will be captured. Finally, the old and new vertex val-

Algorithm 8 Vertex-centric trimming algorithm.

```
 1: function TRIM(Vertex v)
 2:     ▷ Construct the subset of incoming edges
 3:     v^s ← ∅
 4:     for e ∈ INCOMINGEDGES(v) do
 5:         if e.source.level ≤ v.level then
 6:             v^s.insert(e)
 7:         end if
 8:     end for
 9:     incomingSet ← CONSTRUCTSUBSET(v.value, v^s)
10:
11:     ▷ Find safe alternate value
12:     v.newValue ← VERTEXFUNCTION(incomingSet)
13:
14:     ▷ Continue trimming if required
15:     continueTrim ←
16:             SHOULDPROPAGATE(v.value, v.newValue)
17:     if continueTrim = true then
18:         SCHEDULECHILDREN(v)
19:     end if
20:
21:     v.value ← v.newValue
22:     v.UPDATE(incomingSet)
23: end function
```

ues are used to determine whether trimming should be done to the children of the vertex. The algorithm requires algorithmic insights which are provided by the developer using a comparator function (Line 15). Depending upon the result of the function, the immediate children in the dependence trees may or may not be scheduled to be processed.

Since multiple deletions can be present in the same update batch, trimming can be performed in such a way that it starts from the highest level of the dependence tree and gradually moves down, rather than starting at multiple deletion points which may be at different tree levels. This way multiple trimming flows get merged into a single flow.

5.2.4 Trimming for Performance

As shown in Figure 5.6, certain deletions can render the approximate values of the affected vertices far away from their final results, causing the branch loop to take a long time to converge. The trimming technique described in Section 5.2 is automatically applicable in such cases to accelerate convergence. For example, in the case of SSSP (Figure 5.6), since the algorithm is monotonically decreasing, our dependence based trimming will keep trimming vertices when their new values are larger than their old values. In Figure 5.6 (a), after $A \rightarrow B$ is deleted, B's value is reset to `MAX` since the subset of incoming edges used to reexecute the update function at B is empty because C depends on B and thus $C \rightarrow B$ is not considered. This significantly accelerates computation since C's value can only be set to `MAX` as well (due to the influence from B).

5.2.5 Safety and Profitability Arguments

Safety

It is straightforward to see that tagging + resetting provides safety because it resets vertex values conservatively. For the dependence-based trimming, as long as the developer appropriately inserts the dependence-tracking API call into the program, *all* value dependences will be correctly recorded. For monotonic algorithms that use one single incoming edge to compute the value of a vertex, these dependence relationships yield a set of dependence trees. When an edge is deleted, the subtree rooted at the target vertex of the deleted edge thus includes a *complete* set of vertices that are directly or transitively impacted by the deleted edge. Given this set of impacted vertices, we next show that trimming produces a safe (recoverable) approximation.

Theorem 5.2.1. *Trimming based on value dependence trees produces a safe approximation.*

Proof. We prove the safety of trimming by analyzing the set of values used for computing the new approximations and showing that these values themselves are not *unsafe approximations*, that is, they are not over-approximations for monotonic increasing algorithms and under-approximations for monotonic decreasing algorithms. Let us consider the deletion of an edge $a \rightarrow b$, which triggers the trimming process. We prove the theorem by contradiction. Suppose the approximation produced is unsafe and let v be the vertex whose approximate value becomes unsafe after trimming is performed. If we were to back-track how the unsafe approximation got computed during trimming, there must be an earliest point at which an unsafe value was introduced and propagated subsequently to v. Let c be such an earliest vertex. Since c's value was safe prior to the deletion of $a \rightarrow b$ but it became unsafe afterwards, c is dependent on $a \rightarrow b$. This means $b \xrightarrow{LT} c$, and now, $c \xrightarrow{LT} v$.

In this case, prior to the edge deletion, the dependence relationship $b \xrightarrow{LT} c$ must have been captured. When the deletion occurs, the trimming process considers the entire subtree rooted at b in the collected dependence trees, which includes the path from b to c. Our algorithm computes a new value for c if its predecessor's value changes against the monotonic direction. This leads to the following two cases:

Case 1 — The value of c's predecessor indeed changes against monotonicity. In this case, c's value is recomputed. As described earlier, only those incoming values that are not in the subtree rooted at c are considered for computing c's new value, which ensures that all the vertices whose values were (directly or indirectly) computed using c's old value do not participate in this new computation. The incoming values selected for computation must

Graphs	#Edges	#Vertices
Friendster (FT) [40]	2.5B	68.3M
Twitter (TT) [17]	2.0B	52.6M
Twitter (TTW) [72]	1.5B	41.7M
UKDomain (UK) [11]	1.0B	39.5M
LiveJournal (LJ) [5]	69M	4.8M

Table 5.2: Real world input graphs.

have safe approximation themselves because c is the earliest vertex whose approximation became unsafe. Hence, using safe, but fewer, incoming values for c can only lead to a safe approximate value for c due to mononicity (*e.g.*, a value higher than the accurate value in a decreasing monotonic algorithm).

Case 2 — The value of c's predecessor does not change against monotonicity. In this case, c's old approximate value is already safe w.r.t. its predecessor.

Combining Case 1 and 2, it is clear to see that there does not exist any vertex c whose value can become unsafe and flow to v in our algorithm. Simple induction on the structure of the dependence tree would suffice to show any vertex v's value must be safe. □

Profitability

It is easy to see that any *safe* approximate value is at least as good as the initial value. Since the value already carries some amount of computation, use of the value would reuse the computation, thereby reaching the convergence faster than using the initial value.

Algorithm	Issue	vertexFunction
SSWP	Correctness	$v.path \leftarrow \max_{e \in \text{inEdges}(v)}(\min(e.source.path, e.weight))$
CC	Correctness	$v.component \leftarrow \min(v.component, \min_{e \in \text{edges}(v)}(e.other.component))$
BFS	Performance	$v.dist \leftarrow \min_{e \in \text{inEdges}(v)}(e.source.dist + 1)$
SSSP	Performance	$v.path \leftarrow \min_{e \in \text{inEdges}(v)}(e.weight + e.source.path)$

Table 5.3: Various vertex-centric graph algorithms.

Algorithm	shouldPropagate
SSWP	$newValue < oldValue$
CC	$newValue > oldValue$
BFS	$newValue > oldValue$
SSSP	$newValue > oldValue$

Table 5.4: shouldPropagate conditions.

5.3 Evaluation

This section presents a thorough evaluation of KickStarter on real-world graphs.

5.3.1 Implementation

KickStarter was implemented in the ASPIRE [129] distributed graph processing system. The graph vertices are first partitioned across nodes and then processed using a vertex-centric model. The iterative processing incorporates a vertex activation technique using bit-vectors to eliminate redundant computation on vertices whose inputs do not change.

		LJ	UK	TTW	TT	FT
SSWP	RST	7.48-10.16 (8.59)	81.22-112.01 (90.75)	94.18-102.27 (99.28)	170.76-183.11 (176.87)	424.46-542.47 (487.04)
	TAG	11.57-14.71 (13.00)	1.73-62.1 (21.42)	27.38-125.91 (71.26)	262.88-278.42 (270.29)	474.64-550.25 (510.52)
	VAD	3.51-5.5 (4.48)	1.17-1.18 (1.17)	21.54-34.38 (27.55)	66.85-130.84 (75.88)	113.3-413.51 (143.72)
CC	RST	6.43-7.93 (7.19)	133.92-166.33 (148.80)	105.16-111.46 (107.54)	113.92-126.35 (126.35)	212.43-230.26 (221.05)
	TAG	10.98-12.81 (11.86)	170.91-203.54 (183.93)	176.91-201.12 (185.84)	193.77-249.93 (208.90)	331.79-386 (360.34)
	VAD	4.89-5.85 (5.30)	1.81-7.75 (4.37)	31.78-33.24 (32.54)	21.98-22.58 (22.29)	38-39.36 (38.56)

Table 5.5: Trimming for correctness: query processing time (in sec) for SSWP and CC, shown in the form of min-max (average).

		LJ	UK	TTW	TT	FT
SSWP	**TAG**	3.1M-3.2M (3.1M)	8.9K-9.7M (4.1M)	1.1K-29.5M (13.4M)	28.5M-28.6M (28.6M)	49.5M-49.5M (49.5M)
	VAD	20.1K-90.8K (60.8K)	2.9K-93.0K (33.4K)	1.0K-4.5K (2.3K)	2.4K-1.1M (106.4K)	20.7K-13.6M (1.3M)
CC	**TAG**	3.2M-3.2M (3.2M)	25.9M-25.9M (25.9M)	31.3M-31.3M (31.3M)	32.2M-32.2M (32.2M)	52.1M-52.1M (52.1M)
	VAD	1.1K-3.1K (1.9K)	320-1.6K (1.0K)	116-463 (212)	241-463 (344)	294-478 (374)

Table 5.6: Trimming for correctness: # reset vertices for SSWP and CC (the lower the better) in the form of min-max (average).

Updates are batched in an in-memory buffer and not applied until the end of an iteration. Value dependence trees are constructed by maintaining the level information for each vertex along with "downward" pointers to children vertices that allow trimming to quickly walk down a tree. A query is performed after a batch of updates is applied. The query asks for the values of *all* vertices in the updated graph. Different types of edge updates are handled differently by KickStarter: edge deletion removes the edge and schedules the edge target for trimming; and standard treatment is employed for edge additions.

5.3.2 Experimental Setup

We used four monotonic graph algorithms in two categories as shown in Table 5.3 and Table 5.4: SingleSourceWidestPaths (SSWP) and ConnectedComponents (CC) may produce incorrect results upon edge deletions whereas SingleSourceShortestPaths (SSSP) and BreathFirstSearch (BFS) produce correct results with poor performance. VERTEX-FUNCTION shows the algorithms of vertex computation, while SHOULDPROPAGATE reports the termination conditions of the trimming process.

The algorithms were evaluated using five real-world graphs listed in Table 5.2. Like [112], we obtained an initial fixed point and streamed in a set of edge insertions and deletions for the rest of the computation. After 50% of the edges were loaded, the remaining edges were treated as edge additions that were streamed in. Furthermore, edges to be deleted

were selected from the loaded graph with a 0.1 probability; deletion requests were mixed with addition requests in the update stream. In our experiments, we varied both the rate of the update stream and the ratio of deletions vs. additions in the stream to thoroughly evaluate the effects of edge deletions.

All experiments were conducted on a 16-node Amazon EC2 cluster. Each node has 8 cores and 16GB main memory, and runs 64-bit Ubuntu 14.04 kernel 3.13.

Techniques Compared We evaluated KickStarter by comparing the following four versions of the streaming algorithm:

- **TAG** uses the *tagging + resetting* based trimming (Section 5.2.2).
- **VAD** uses our *value dependence* based trimming (Section 5.2.3).
- **TOR** implements Tornado [112]. This approach is used as the baseline for BFS and SSSP, as **TOR** generates correct results for these algorithms.
- **RST** does not perform trimming at all and instead *resets* values of all vertices. This technique serves as the baseline for SSWP and CC because TOR does not generate correct results for them (as already shown in Figure 5.5 in Section 5.1).

To ensure a fair comparison among the above versions, queries were generated in such a way that each query had the same number of pending edge updates to be processed. Unless otherwise stated, 100K updates with 30% deletions were applied before the processing of each query.

5.3.3 Trimming for Correctness

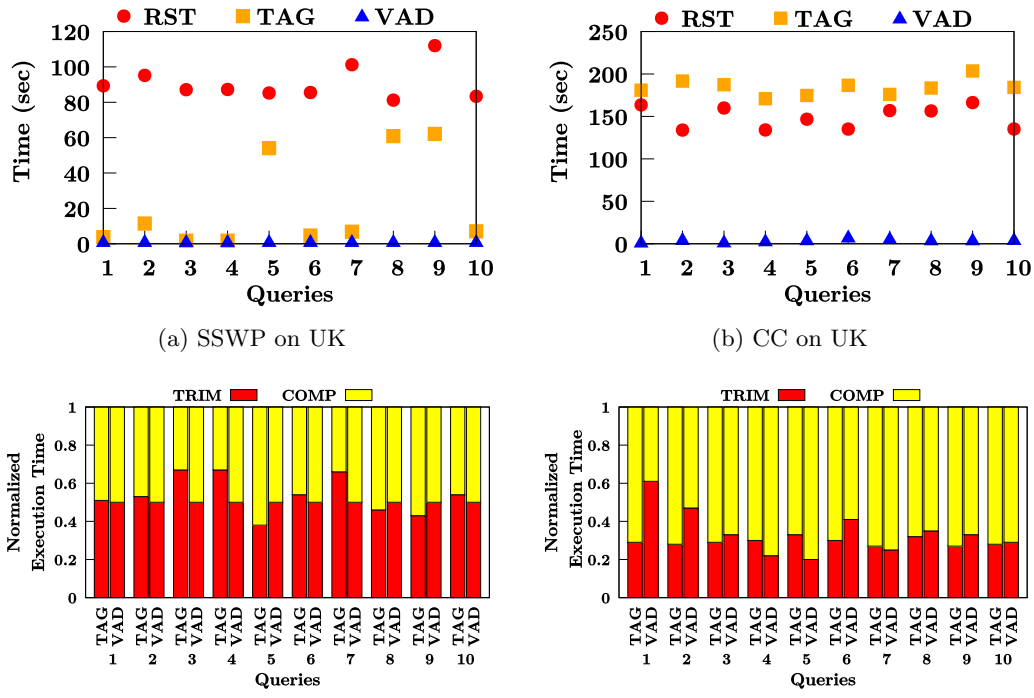
We first study the performance of our trimming techniques, TAG and VAD, to generate correct results for SSWP and CC — Table 5.5 shows the average, minimum, and maximum execution times (in seconds) to compute the query results using TAG, VAD, and RST (baseline).

We observe that VAD consistently outperforms RST. On average, VAD for SSWP and CC performs 17.7× and 10× faster than RST, respectively. These significant speedups were achieved due to the incremental processing in VAD that maximizes the use of computed approximate values, as can be seen in Table 5.6 — only a subset of vertices have to discard their approximate values (via resetting). Since RST discards the entire approximation upon edge deletions (*i.e.*, resetting all impacted vertices), computation at every tagged vertex starts from scratch, resulting in degraded performance.

Finally, for the UK graph, VAD performs noticeably better for SSWP than for CC mainly because safe approximate values computed for SSWP were closer to the final solution than those for CC. The reason is as follows. For CC, if the component ID for a vertex in a component changes, this change is likely to get propagated to all the other vertices in that component. This means that when trimming finds a safe approximate value, the value may still be changed frequently during the forward execution. For SSWP, on the other hand, if the path value for a vertex changes, the change does not affect many other vertices. Hence, only small local changes may occur in vertex values before the computation converges. As a result, SSWP took less time than CC to finish the branch loop (less than a second in all cases for SSWP vs. between 1 and 4 seconds for CC).

Next, we compare the tagging+resetting algorithm TAG with VAD and RST. In most cases, TAG outperforms RST. However, TAG performs worse than RST for CC because the overhead of using TAG outweighs the benefit provided by trimming — due to TAG's conservative nature, a very large portion of the graph is tagged and their values are reset. This can be seen in Table 5.6 where there are millions of vertices whose values are reset in CC.

Under TAG, the number of vertices whose values are reset is significantly higher than that under VAD (see Table 5.6). Hence, VAD consistently outperforms TAG. Note that the reason why VAD works well for CC is that since CC propagates component IDs, many neighbors of a vertex may have the same component ID and thus trimming based



(a) SSWP on UK

(b) CC on UK

(c) SSWP on UK

(d) CC on UK

Figure 5.8: Time taken to answer queries.

on value dependence may have more approximate values to choose from. As a result, VAD resets far fewer vertices than TAG.

Figures 5.8a and 5.8b show the performance of RST, TAG, and VAD for the first 10 queries for SSWP and CC on UK. The performance of TAG is more sensitive to edge deletions than VAD – for SSWP, while the solutions for many queries were computed quickly by TAG, some queries took significantly longer processing time. VAD, on the other hand, is less sensitive to edge deletions because it is able to attribute the effects of the deletions to a smaller subset of vertices.

		LJ	UK	TTW	TT	FT
SSSP	**TOR**	1.27-102.88 (39.10)	2.84-119.03 (24.90)	17.62-131.9 (112.57)	42.13-584.64 (190.78)	90.59-179.83 (163.99)
	TAG	3.25-4.49 (3.97)	2.03-2.94 (2.19)	46.06-52.5 (48.96)	98.59-118.23 (105.73)	131.22-150.16 (142.60)
	VAD	2.12-3.22 (2.55)	1.33-1.5 (1.41)	28.68-32.33 (30.21)	41.35-48.65 (44.19)	93.74-101.67 (97.22)
BFS	**TOR**	1.17-77.05 (7.17)	1.24-588.09 (142.55)	23.94-1015.76 (199.23)	55-283.71 (120.45)	190.52-2032.38 (881.17)
	TAG	3.47-4.43 (3.88)	1.81-5.14 (1.97)	51.08-58.3 (54.36)	110.75-192.71 (127.54)	143.21-334.07 (166.60)
	VAD	1.96-3.37 (2.59)	1.21-3.88 (1.42)	32.02-34.86 (32.96)	69.43-91.88 (74.27)	107.4-136.73 (114.56)

Table 5.7: Trimming for performance: query processing times (in sec) for SSSP and BFS in the form: min-max (average).

Finally, Figures 5.8c and 5.8d compare the performance of the two phases of the branch loop execution: *trimming* (TRIM) and *computation* (COMP). Since CC is more sensitive to edge additions and deletions compared to SSWP, it took longer processing time to converge to the correct result. Hence, for CC, the percentages of the time spent by both TAG and VAD on the trimming phase are lower than those on the computation phase. SSWP, on the other hand, needs less time to converge because the approximate values available for incremental processing are closer to the final values; hence, the time taken by the trimming phase becomes comparable to that taken by the computation phase.

		LJ	UK	TTW	TT	FT
SSSP	**TAG**	8.2K-59.8K (25.9K)	4.1K-193.4K (36.4K)	19.7K-183.7K (89.4K)	6.2K-196.7K (51.5K)	19.8K-31.2K (25.4K)
	VAD	1.7K-40.1K (7.0K)	2.9K-52.2K (16.6K)	2.1K-77.7K (19.6K)	836-110.9K (11.1K)	4.5K-12.5K (8.0K)
BFS	**TAG**	10.8K-354.5K (79.0K)	1.3K-483.0K (35.5K)	20.9K-1.2M (457.6K)	44.2K-8.6M (1.1M)	19.1K-4.5M (469.8K)
	VAD	5.5K-116.6K (36.4K)	3.2K-469.9K (41.2K)	860-3.1K (1.6K)	742-1.4K (1.1K)	2.7K-5.2K (3.4K)

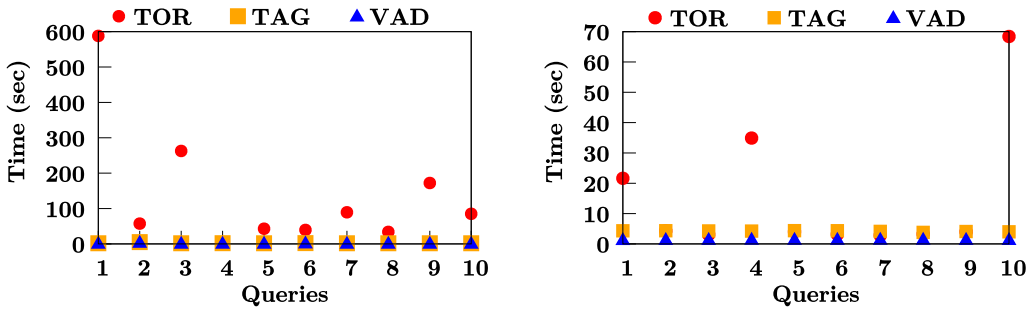
Table 5.8: Trimming for performance: number of reset vertices for SSSP and BFS in the form: min-max (average).

Nevertheless, as Table 5.5 shows, the trimming phase has very little influence on the overall processing time due to its lightweight design and parallel implementation.

5.3.4 Trimming for Performance

This set of experiments help us understand how different trimming mechanisms can improve the performance of query processing for BFS and SSSP. For these two algorithms, as explained in Section 5.1, although the baseline TOR (Tornado) produces correct results, it can face performance issues. Table 5.7 shows the average, minimum, and maximum execution times (in seconds) to compute the query results for SSSP and BFS by TOR, TAG, and VAD.

VAD consistently outperforms TOR. For example, VAD for SSSP and BFS are



(a) SSSP on UK

(b) BFS on UK

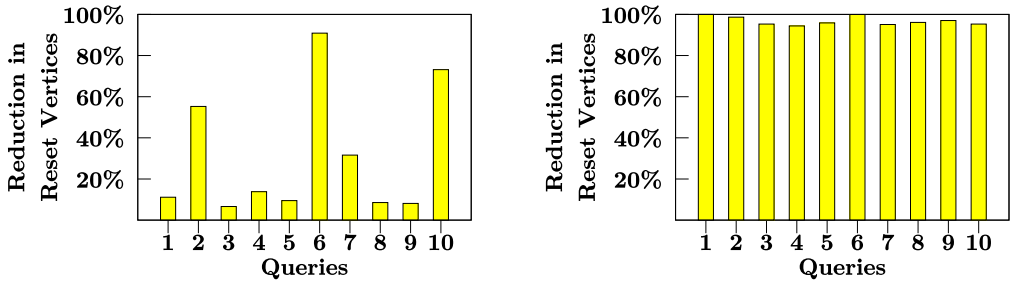
Figure 5.9: Trimming for performance: time taken to compute answer queries by TAG and VAD.

overall 23.7× and 8.5× faster than TOR, respectively. Figure 5.9a and Figure 5.9b show the performance for answering the first 10 queries for SSSP and BFS. Since TOR leverages incremental processing, its performance for some queries is competitive with that of VAD. However, different edge deletions impact the approximation differently and in many cases, TOR takes a long time to converge, leading to degraded performance.

While TAG consistently outperforms TOR, its conservative resetting of a larger set of vertex values (as seen in Table 5.8) introduces overhead that reduces its overall benefit.

5.3.5 Effectiveness of the Trimmed Approximation

To understand whether the new approximate values computed by trimming are beneficial, we compare VAD with a slightly modified version VAD-Reset that does not compute new approximate values. This version still identifies the set of impacted vertices using dependence tracking, but simply resets the values of all vertices in the set. Figure 5.10 shows the reductions in the numbers of reset vertices achieved by VAD over VAD-Reset. The higher the reduction, the greater is the computation reused. This comparison was done



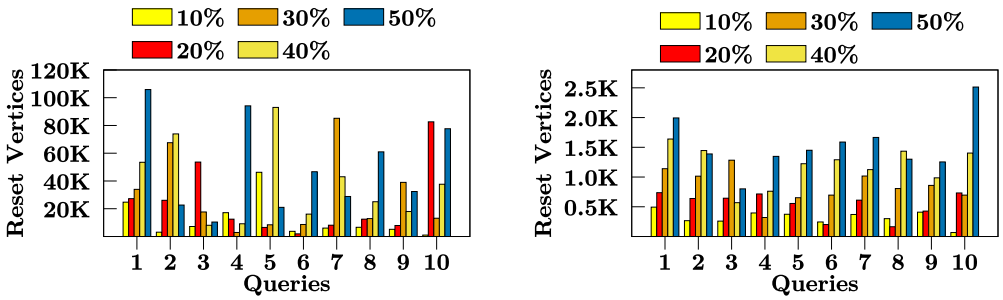
(a) SSWP on UK

(b) CC on UK

Figure 5.10: Reduction in # of vertices reset by VAD compared to VAD-Reset.

on the first 10 queries for SSWP and CC over the UK graph. We observe that the reduction varies significantly between SSWP and CC. This is mainly due to the different shapes of the dependence trees constructed for CC and SSWP. For CC, the dependence trees are fat (*i.e.*, vertices have more children) and, hence, if a vertex's value is reset, the trimming process needs to continue resetting many of its children vertices, hurting performance significantly. In fact, 5.7K-25.9M vertices were reset for CC under VAD-Reset.

As CC propagates component IDs and the IDs in a vertex's neighborhood are often the same (because the vertices in a neighborhood likely belong to the same component), good approximate values are often available under VAD, which greatly reduces the number of reset vertices (to 320-1.3K). For SSWP, on the other hand, its dependence trees are thinner (*i.e.*, vertices have less children), and hence VAD-Reset does a reasonably good job as well, resetting only 3.4K-151K vertices. This number gets further reduced to 2.9K-85.2K when VAD is used. We have also observed that the benefits achieved for SSWP vary significantly across different queries; this is due to the varying impacts of deletions that affect different regions of dependence trees and thus availability of safe approximate values.



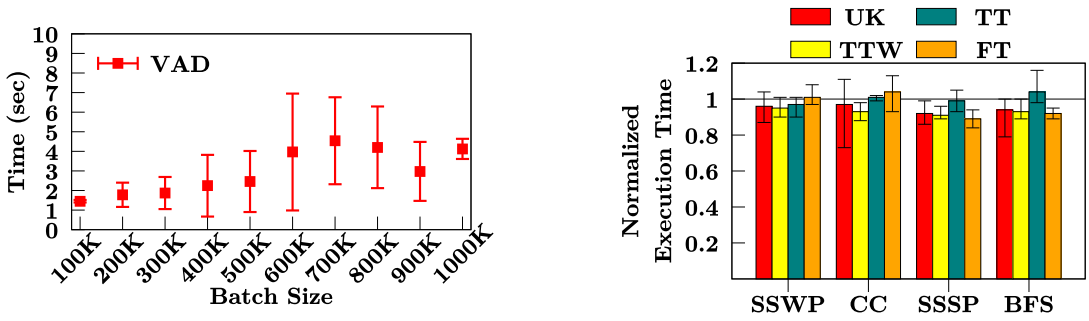
(a) SSWP on UK

(b) CC on UK

Figure 5.11: Numbers of reset vertices with different deletion percentages in the batch.

5.3.6 Sensitivity to Edge Deletions & Batch Size

We study the sensitivity of the trimming effectiveness to the number of deletions performed in the update stream. In Figure 5.11, we vary the percentage of deletions in an update batch from 10% to 50% while maintaining the same batch size of 100K edge updates. While the trend varies across different queries and algorithms, it is important to note that our technique found safe approximations in many cases, keeping the number of reset values low even when the number of deletions increases.



(a) Time taken by queries with varying batch size for SSWP on UK

(b) Dependence tracking overhead

Figure 5.12: Query time and dependence tracking overhead.

In Figure 5.12a, we varied the number of edge updates applied for each query from 100K to 1M while setting the deletion percentage to 30%. Clearly, the increase in the number of edge updates, and hence in the number of edge deletions, do not have much impact on the performance of the forked branch loop. This is mainly because the number of reset vertices remains low — it increases gradually from 31K to 230K.

5.3.7 Dependence Tracking Overhead

Finally, we study the overhead of our dependence tracking technique by measuring the performance of the system with only edge additions. Since the handling of edge additions does not need trimming, the difference of the running time between VAD and TOR is the tracking overhead. Figure 5.12b shows the overall execution times for queries under VAD normalized *w.r.t.* TOR with only the edge addition requests. That is, the deletion percentage is set to 0%. In this case, all the four algorithms leverage the incremental processing in TOR and hence the maintained approximation is never changed. The error bars in Figure 5.12b indicate the min and max values to help us understand the performance variations. The overall performance is only slightly influenced (max overhead bars are slightly above 1 in most cases) and the query answering time under VAD increases by 13%.

5.4 Summary

In this chapter, we exploited the algorithmic asynchrony to efficiently process streaming graphs. To leverage from computed results, we developed a dynamic dependence based incremental processing technique that quickly identifies the minimal set of vertices to be trimmed so that the results become safe and profitable. Furthermore, we trim the unsafe values by adjusting them instead of resetting them to initial values, hence making use of previous computed values. KickStarter quickly computes query results while simultaneously providing 100% correctness guarantees in presence of edge deletions.

Chapter 6

Out-of-core Processing

So far in the thesis we studied how the asynchronous model can be used to develop various runtime techniques and optimizations to process static and dynamic graphs in a distributed setting. While these techniques can be further specialized to perform different kinds of analyses over a cluster, in this chapter we further demonstrate the efficacy of asynchronous model across processing environments beyond a distributed setting. We choose the popular *out-of-core* processing environment that employs disks for processing large amounts of data due to main memory constraints. In such a setting, we identify key opportunities to leverage the algorithmic asynchrony so that large scale graph processing can be further improved.

Out-of-core graph systems can be classified into two major categories based on their computation styles: *vertex-centric* and *edge-centric*. At the heart of both types of systems is a well-designed, disk-based partition structure, along with an efficient *iterative, out-of-core* algorithm that accesses the partition structure to load and process a small

portion of the graph at a time and write updates back to disk before proceeding to the next portion. As an example, GraphChi [73] uses a *shard* data structure to represent a graph partition: the graph is split into multiple shards before processing; each shard contains edges whose target vertices belong to the same logical interval. X-Stream [103] partitions vertices into *streaming partitions*. GridGraph [153] constructs 2-dimensional edge blocks to minimize I/O.

Despite much effort to exploit locality in the partition design, existing systems use *static partition layouts*, which are determined before graph processing starts. In every single computational iteration, each partition is loaded entirely into memory, although a large number of edges in the partition are not strictly needed.

Consider an iteration in which the values for only a small subset of vertices are changed. Such iterations are very common when the computation is closer to convergence and values for many vertices have already stabilized. For vertices that are not updated, their values do not need to be pushed along their outgoing edges. Hence, the values associated with these edges remain the same. The processing of such edges (*e.g.*, loading them and reading their values) would be completely redundant in the next iteration because they make zero *new contribution* to the values of their respective target vertices.

Repeatedly loading these edges creates significant I/O inefficiencies, which impacts the overall graph processing performance. This is because data loading often takes a major portion of the graph processing time. As an example, over 50% of the execution time for PageRank is spent on partition loading, and this percentage increases further with the size of the input graph (Section 6.1).

Key Idea

We aim to reduce the above I/O inefficiency in out-of-core graph systems by exploring the idea of *dynamic partitions* that are created by omitting the edges that are not updated.

While our idea is applicable to *all* disk-based systems, in this work we focus on dynamically adjusting the *shard structure* used in GraphChi. We choose GraphChi as the starting point because: (1) it is a representative of extensively-used vertex-centric computation; (2) it is under active support and there are a large number of graph programs already implemented in it; and (3) its key algorithm has been incorporated into *GraphLab Create* [47], a commercial product of Dato, which performs both distributed and out-of-core processing. Hence, the goal of this work is *not* to produce a brand new system that is faster than all existing graph systems, but instead, to show the *generality and effectiveness* of our optimization, which can be implemented in other systems as well.

Challenges

Using dynamic partitions requires much more than recognizing unnecessary edges and removing them. There are two main technical challenges that need to be overcome.

The first challenge is how to perform vertex computation in the presence of missing edges that are eliminated during the creation of a dynamic partition. Although these edges make no impact on the forward computation, current programming/execution models all assume the presence of all edges of a vertex to perform value updates. To solve the problem, we begin with proposing a delay-based computation model (Section 6.2) that *delays* the computation of a vertex with a missing edge until a special *shadow iteration* in which all edges are brought into memory from static partitions.

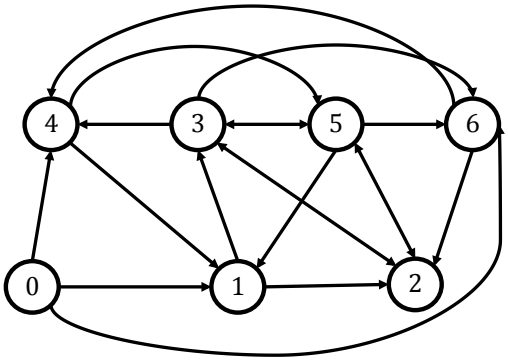
Since delays introduce overhead, to reduce delays, we further propose an *accumulation-based* programming/execution model (Section 6.3) that enables *incremental vertex computation* by expressing computation in terms of contribution increments flowing through edges. As a result, vertices that *only have missing incoming edges* can be processed instantly without needing to be delayed because the increments from missing incoming edges are guaranteed to be zero. Computation for vertices with missing outgoing edges will still be delayed, but the number of such vertices is often very small.

The second challenge is how to efficiently build partitions on the fly. Changing partitions during processing incurs runtime overhead; doing so frequently would potentially make overheads outweigh benefits. We propose an additional optimization (Section 6.4) that constructs dynamic partitions only during shadow iterations. We show, theoretically (Section 6.4) and empirically (Section 6.5), that this optimization leads to I/O reductions rather than overheads. Our experiments with five common graph applications over six real graphs demonstrate that using dynamic shards in GraphChi accelerates the overall processing by up to 2.8× (on average 1.8×). While the accelerated version is still slower than X-Stream in many cases (Section 6.5.3), this performance gap is reduced by 40% after dynamic partitions are used.

6.1 The Case for Dynamic Partitions

Background

A graph $G = (V, E)$ consists of a set of vertices, V, and a set of edges E. The vertices are numbered from 0 to $|V| - 1$. Each edge is a pair of the form $e = (u, v)$, $u, v \in V$. u is



(a) Example graph.

Shard 0

Src	Dst	Value
0	1	e_0
1	2	e_1
3	2	e_2
4	1	e_3
5	1	e_4
	2	e_5
6	2	e_6

Shard 1

Src	Dst	Value
0	4	e_7
1	3	e_8
2	3	e_9
	5	e_{10}
3	4	e_{11}
	5	e_{12}
4	5	e_{13}
5	3	e_{14}
6	4	e_{15}

Shard 2

Src	Dst	Value
0	6	e_{16}
3	6	e_{17}
5	6	e_{18}

(b) Shards representation.

Figure 6.1: An example graph partitioned into shards.

the source vertex of e and v is e's destination vertex. e is an incoming edge for v and an outgoing edge for u. The vertex-centric computation model associates a data value with each edge and each vertex; at each vertex, the computation retrieves the values from its incoming edges, invokes an update function on these values to produce the new vertex value, and pushes this value out along its outgoing edges.

The goal of the computation is to "iterate around" vertices to update their values until a global "fixed-point" is reached. There are many programming models developed to support vertex-centric computation, of which the `gather-apply-scatter` (GAS) model is perhaps the most popular one. We will describe the GAS model and how it is adapted to work with dynamic shards in §6.3. A vertex-centric system iterates around vertices to update their values until a global "fixed-point" is reached.

In GraphChi, the IDs of vertices are split into n disjoint logical intervals, each of which defines a shard. Each shard contains all edge entries whose *target vertices* belong to its defining interval. In other words, the shard only contains incoming edges of the vertices in the interval. As an illustration, given the graph shown in Figure 6.1a, the distribution of

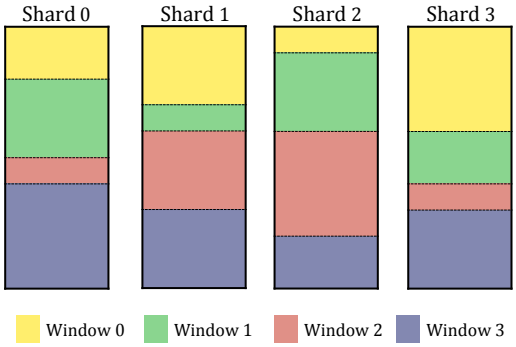
its edges across three shards is shown in Figure 6.1b where vertices 0–2, 3–5, and 6 are the three intervals that define the shards. If the source of an edge is the same as the previous edge, the edge's *src* field is empty. The goal of such a design is to reduce disk I/O by maximizing sequential disk accesses.

Vertex-centric computation requires the presence of all (in and out) edges of a vertex to be in memory when the update is performed on the vertex. Since edges of a vertex may scatter to different shards, GraphChi uses an efficient parallel sliding window (PSW) algorithm to minimize random disk accesses while loading edges. First, edges in a shard s are sorted on their source vertex IDs. This enables an important property: while edges in s can come out of vertices from different intervals, those whose sources are in the same interval i are located contiguously in the shard defined by i.

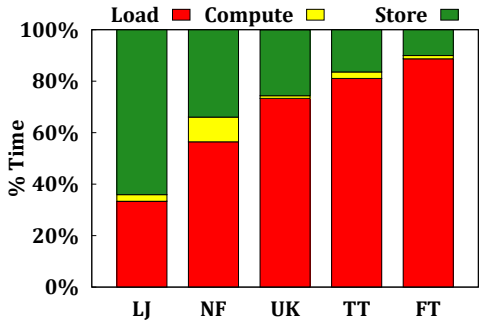
When vertices v in the interval of s are processed, GraphChi only needs to load s (*i.e.*, memory shard, containing all v's incoming edges and part of v's outgoing edges) and a small block of edges from each other shard (*i.e.*, sliding shard, containing the rest of v's outgoing edges) – this brings into memory a *complete* set of edges for vertices belonging to the interval. Figure 6.2a illustrates GraphChi's edge blocks. The four colors are used, respectively, to mark the blocks of edges in each shard whose sources belong to the four intervals defining these shards.

Motivation

While the PSW algorithm leverages disk locality, it suffers from redundancy. During computation, a shard contains edges both with and without updated values. Loading the entire

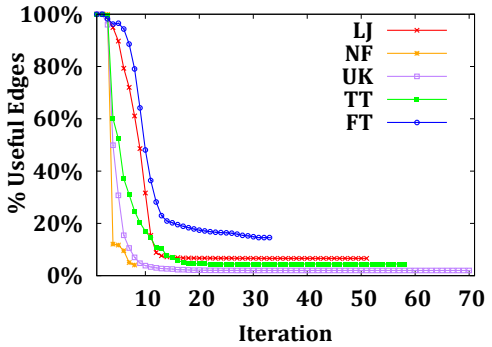


(a) Shards and sliding windows used in GraphChi.

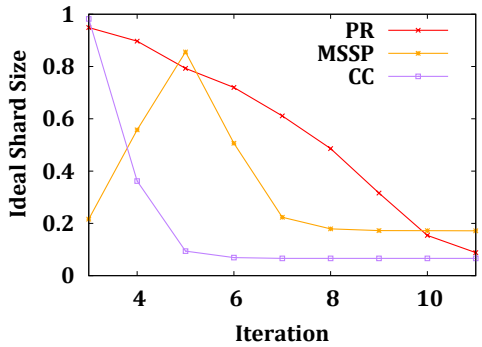


(b) Breakdown of PageRank execution time into load, compute, and store times.

Figure 6.2: An illustration of sliding windows and the PageRank execution statistics.



(a) Percentages of updated edges across iterations for the PageRank algorithm.



(b) Ideal shard sizes normalized w.r.t. the static shard size for LJ input graph.

Figure 6.3: Useful data in static shards.

shard in every iteration involves wasteful effort of loading and processing edges that are guaranteed to make zero new contribution to the value computation. This effort is significant because (1) the majority of the graph processing cost comes from the loading phase, and (2) at the end of each iteration, there are a large number of edges whose values are unchanged. Figure 6.2b shows a breakdown of the execution times of PageRank in GraphChi for five real graphs, from the smallest *LiveJournal* (LJ) with 69M edges to *Friendster* (FT) with 2.6B edges. Further details for these input graphs can be found in Table 6.4.

In these experiments, the I/O bandwidth was fully utilized. Note that the data loading cost increases as the graph becomes larger – for *Friendster*, data loading contributes to over 85% of the total graph processing time. To understand if the impact of data loading is pervasive, we have also experimented with X-Stream [103]. Our results show that the *scatter* phase in X-Stream, which streams all edges in from disk, takes over 70% of the total processing time for PageRank on these five graphs.

To understand how many edges contain necessary data, we calculate the percentages of edges that have updated values across iterations. These percentages are shown in Figure 6.3a. The percentage of updated edges drops significantly as the computation progresses and becomes very low when the execution comes close to convergence. Significant I/O reductions can be expected if edges not updated in an iteration are completely eliminated from a shard and not loaded in the next iteration.

Figure 6.3b illustrates, for three applications PageRank (PR), MultipleSourceShortestPath (MSSP), and ConnectedComponents (CC), how the size of an *ideal* shard changes as computation progresses when the LiveJournal graph is processed. In each iteration, an ideal shard only contains edges that have updated values from the previous iteration. Observe that it is difficult to find a one-size-fits-all static partitioning because, for different algorithms, when and where useful data is produced changes dramatically, and thus different shards are needed.

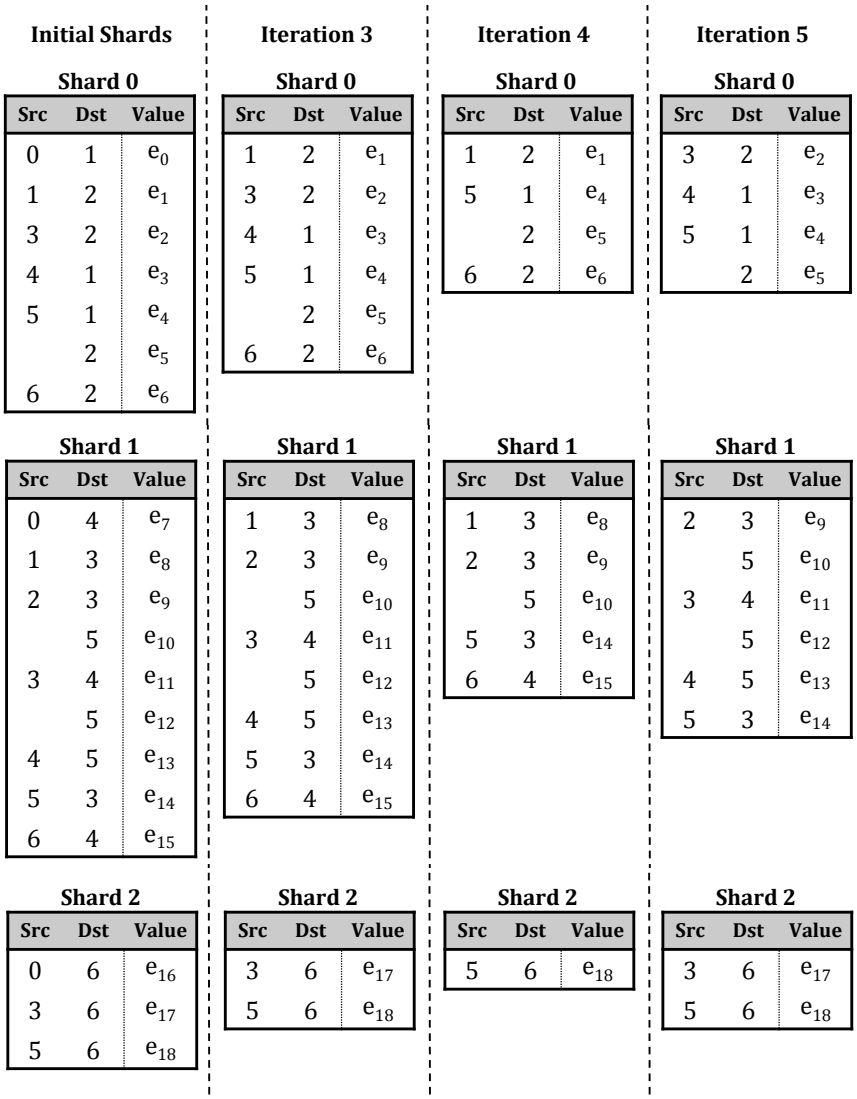


Figure 6.4: Dynamic shards for the example graph in Figure 6.1a created for iteration 3, 4 and 5.

Overview of Techniques

The above observations strongly motivate the need for *dynamic shards* whose layouts can be adapted. Conceptually, for each static shard s and each iteration i in which s is processed, there exists a dynamic shard d_i that contains a subset of edges from s whose values are updated in i. Figure 6.4 shows the dynamic shards created for Iteration 3, 4 and 5 during

the processing of the example graph shown in Figure 6.1a. After the 2^{nd} iteration, vertex 0 becomes inactive, and hence, its outgoing edges to 4 and 6 are eliminated from the dynamic shards for the 3^{rd} iteration. Similarly, after the 3^{rd} iteration, the vertices 3 and 4 become inactive, and hence, their outgoing edges are eliminated from the shards for Iteration 4. In Iteration 4, the three shards contain only 10 out of a total of 19 edges. Since loading these 10 edges involves much less I/O than loading the static shards, significant performance improvement can be expected. To realize the benefits of dynamic shards by reducing I/O costs, we have developed three techniques:

(1) Processing Dynamic Shards with Delays – Dynamic shards are iteratively processed like static shards; however, due to missing edges in a dynamic shard, we may have to delay computation of vertices. We propose a delay based shard processing algorithm that places delayed vertices in an in-memory buffer and periodically performs *shadow iterations* that process the delayed requests by bringing in memory all edges for delayed vertices.

(2) Programming Model for Accumulation-Based Computation – Delaying the computation of a vertex if any of its edge is missing can slow the progress of the algorithm. To overcome this challenge we propose an *accumulation-based* programming model that expresses computation in terms of incremental contributions flowing through edges. This maximizes the processing of a vertex by allowing incremental computations to be performed using available edges and thus minimizes the impact of missing edges.

(3) Optimizing Shard Creation – Finally, we develop a practical strategy for balancing the cost of creating dynamic shards with their benefit from reduced I/O by adapting the frequency of shard creation and controlling when a shadow iteration is triggered.

6.2 Processing Dynamic Shards with Delays

Although dynamic shard provides a promising solution to eliminating redundant loading, an immediate question is how to compute vertex values when edges are missing. To illustrate, consider the following graph edges: $u \to v \to w$. Suppose in one iteration the value of v is not changed, which means v becomes inactive and the edge $v \to w$ is not included in the dynamic shard created for the next iteration. However, the edge $u \to v$ is still included because a new value is computed for u and pushed out through the edge. This value will be reaching v in the next iteration. In the next iteration, the value of v changes as it receives the new contribution from $u \to v$. The updated value of v then needs to be pushed out through the edge $v \to w$, which is, however, not present in memory.

To handle missing edges, we allow a vertex to *delay its computation* if it has a missing edge. The delayed computations are batched together and performed in a special periodically-scheduled iteration called *shadow iteration* where all the (in- and out-) edges of the delayed vertices are brought in memory. We begin by discussing dynamic shard creation and then discuss the handling of missing edges.

Creating Dynamic Shards

Each computational iteration in GraphChi is divided into three phases: load, compute, and write-back. We build dynamic shards at the end of the compute phase but before write-back starts. In the compute phase, we track the set of edges that receive new values from their source vertices using a *dirty* mark. During write-back, these dirty edges are written into new shards to be used in the next iteration. Evolving graphs can be supported by marking the dynamically added edges to be dirty and writing them into new dynamic shards. The

shard structure has two main properties contributing to the minimization of random disk accesses: (1) *disjoint edge partitioning* across shards and (2) *ordering of edges* based on source vertex IDs inside each shard. Dynamic shards also follow these two properties: since we do not change the logical intervals defined by static partitioning, the edge disjointness and ordering properties are preserved in the newly generated shards. In other words, for each static shard, we generate a dynamic shard, which contains a subset of edges that are stored in the same order as in the static shard. Although our algorithm is inexpensive, creating dynamic shards for every iteration incurs much time overhead and consumes large disk space. We will discuss an optimization in Section 6.4 that can effectively reduce the cost of shard creation.

Processing Dynamic Shards

Dynamic shards can be iteratively processed by invoking the user-defined update function on vertices. Although a dynamic shard contains fewer edges than its static counterpart, the logical interval to which the shard belongs is not changed, that is, the numbers of vertices to be updated when a dynamic shard and its corresponding static shard are processed are the same. However, when a dynamic shard is loaded, it contains only *subset* of edges for vertices in its logical interval. To overcome this challenge, we *delay* the computation of a vertex if it has a missing (incoming or outgoing) edge. The delayed vertices are placed in an in-memory delay buffer. We periodically process these delayed requests by bringing in memory all the incoming and outgoing edges for vertices in the buffer. This is done in a special *shadow iteration* where static shards are also loaded and updated.

Algorithm 9 Algorithm for a shadow iteration.

$S = \{S_0, S_1, ..., S_{n-1}\}$: set of n static shards
$DS^i = \{DS_0^i, DS_1^i, ..., DS_{n-1}^i\}$: set of n dynamic shards for Iteration i
$DS = [DS^0, DS^1, ...]$: vector of dynamic shard sets for Iteration 0, 1, ...
V_i: set of vertex IDs belonging to Interval i
DB: delay buffer containing IDs of the delayed vertices
lastShadow: ID of the last shadow iteration
function SHADOW-PROCESSING(Iteration *ite*)
    for each Interval k from 0 to n **do**
        LOAD-ALL-SHARDS(ite, k)
        parallel-for Vertex $v \in DB \cap V_k$ **do**
            UPDATE(v) //user-defined vertex function
        end parallel-for
        produce S'_k by writing updates to the static shard S_k
        create a dynamic shard DS_k^{ite} for the next iteration
    end for
    remove $DS^{lastShadow} \dots DS^{ite-1}$
    lastShadow $\leftarrow$ *ite*
    clear the delay buffer DB
end function

function LOAD-ALL-SHARDS(Iteration *ite*, Interval j)
    LOAD-MEMORY-SHARD(S_j)
    parallel-for Interval $k \in [0, n]$ **do**
        if $k \neq j$ **then**
            LOAD-SLIDING-SHARD(S_k)
        end if
    end parallel-for
    for each Iteration k from *lastShadow* to $ite - 1$ **do** =
        LOAD-MEMORY-SHARD-AND-OVERWRITE(DS_j^k)
        parallel-for Interval $i \in [0, n]$ **do**
            if $i \neq j$ **then**
                LOAD-SLIDING-SHARD-AND-OVERWRITE(DS_i^k)
            end if
        end parallel-for
        if $k = ite - 1$ **then**
            MARK-DIRTY-EDGES( )
        end if
    end for
end function

Since a normal iteration has similar semantics as those of iterations in GraphChi, we refer the interested reader to [73] for its details. Here we focus our discussion on shadow iterations. The algorithm of a shadow iteration is shown in Algorithm 9. A key feature of this algorithm is that it loads the static shard (constructed during pre-processing) to which each vertex in the delay buffer belongs to bring into memory all of its incoming and outgoing edges for the vertex computation. This is done by function LOAD-ALL-SHARDS shown in Lines 21–39 (invoked at Line 9).

However, only loading static shards would not solve the problem because they contain out-of-date data for edges that have been updated recently. The most recent data are scattered in the dynamic shards $DS^{lastShadow} \dots DS^{ite-1}$ where *lastShadow* is the ID of the last shadow iteration and *ite* is the ID of the current iteration. As an example, consider Shard 0 in Figure 6.4. At the end of iteration 5, the most recent data for the edges $1 \rightarrow 2$, $3 \rightarrow 2$, and $0 \rightarrow 1$ are in DS_0^4, DS_0^5, and S_0, respectively, where DS_j^i represents the dynamic shard for interval j created for iteration i, and S_j denotes the static shard for interval j.

To guarantee that most recent updates are retrieved in a shadow iteration, for each interval j, we sequentially load its static shard S_j (Line 22) and dynamic shards created since the last shadow iteration $DS^{lastShadow} \dots DS^{ite-1}$ (Line 29), and let the data loaded later *overwrite* the data loaded earlier for the same edges. LOAD-ALL-SHARDS implements GraphChi's PSW algorithm by loading (static and dynamic) memory shards entirely into memory (Lines 22 and 29) and a sliding window of edge blocks from other (static and dynamic) shards (Lines 23–27 and 30–34). If k becomes the ID of the iteration right before

the shadow iteration (Lines 35–37), we mark dirty edges to create new dynamic shards for the next iteration (Line 14).

After the loop at Line 8 terminates, we remove all intermediate dynamic shards (Line 16) and set *lastShadow* to *ite* (Line 17). These shards are not needed, because the static shards are already updated with the most recent values in this iteration (Line 13). One can view static shards as "checkpoints" of the computation and dynamic shards as intermediate "increments" to most recent checkpoint. Finally, the delay buffer is cleared.

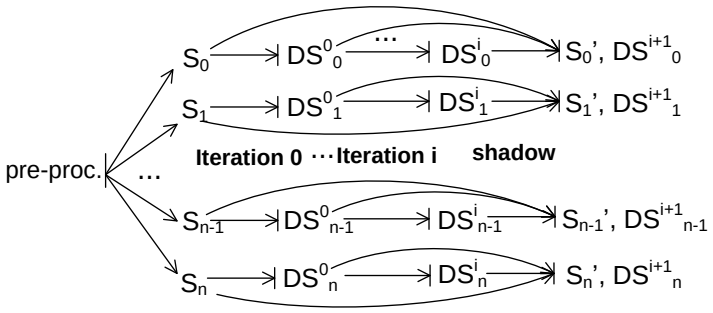


Figure 6.5: Processing using dynamic shards.

Figure 6.5 illustrates the input and output of each computational iteration. Static shards $S_0 \ldots S_n$ are statically constructed. Each regular iteration i produces a set of dynamic shards $DS_0^i \ldots DS_n^i$, which are fed to the next iteration. A shadow iteration loads all static shards and intermediate dynamic shards, and produces (1) updated static shards $S_0' \ldots S_n'$ and (2) new dynamic shards $DS_0^{i+1} \ldots DS_n^{i+1}$ to be used for the next iteration.

It may appear that the delay buffer can contain many vertices and consume a lot of memory. However, since the amount of memory needed to represent an incoming edge is higher than that to record a vertex, processing dynamic shards with the delay buffer is actually more memory-efficient than processing static shards where all edges are available.

Delaying a vertex computation when *any* of its edge is missing can cause too many vertices to be delayed and negatively impact the the computation progress. For example, when running PageRank on *UKDomain*, *Twitter*, and *Friendster* graphs, immediately after the dynamic shards are created, 64%, 70%, and 73% of active vertices are delayed due to at least one missing incoming or outgoing edge. Frequently running shadow iterations may get data updated quickly at the cost of extra overhead, while doing so infrequently would reduce overhead but slow down the convergence. Hence, along with dynamically capturing the set of edges which reflect change in values, it is important to modify the computation model so that it maximizes computation performed using available values.

Section 6.3 presents an optimization for our delay-based computation to limit the number of delayed computations. The optimization allows a common class of graph algorithms to perform vertex computation if a vertex only has missing incoming edges. While the computation for vertices with missing outgoing edges still need to be delayed, the number of such vertices is much smaller, leading to significantly reduced delay overhead.

6.3 Accumulation-based Computation

This section presents an *accumulation-based* programming/execution model that expresses computation in terms of *incremental contributions* flowing through edges. Our insight is that if a vertex is missing an incoming edge, then the edge is guaranteed to provide zero *new contribution* to the vertex value. If we can design a new model that performs updates based on *contribution increments* instead of actual contributions, the missing incoming edge can be automatically treated as *zero* increment and the vertex computation can be performed without delay.

We discuss our approach based on the popular Gather-Apply-Scatter (GAS) programming model [73, 85, 44] where vertex computation is divided in three distinct phases: the `gather` phase reads incoming edges and produces an aggregated value using a user-defined aggregation function; this value is fed to the `apply` phase to compute a new value for a vertex; in the `scatter` phase, the value is propagated along the outgoing edges.

6.3.1 Programming Model

Our accumulation-based model works for a common class of graph algorithms whose GAS computation is *distributive* over aggregation. The user needs to program the GAS functions in a slightly different way to propagate *changes in values* instead of actual values. In other words, the semantics of vertex data remains the same while data on each edge now encodes the delta between the old and the new value of its source vertex. This semantic modification relaxes the requirement that all incoming edges of a vertex have to be present to perform vertex computation.

The new computation semantics requires minor changes to the GAS programming model. (1) `Extract` the gathered value using the old vertex value. This step is essentially an inverse of the `apply` phase that uses its output (*i.e.*, vertex value) to compute its input (*i.e.*, aggregated value). (2) `Gather` edge data (*i.e.*, from present incoming edges) and aggregate it together with the output of `extract`. Since this output represents the contributions of the previously encountered incoming edges, this step incrementally adds new contributions from the present incoming edges to the old contributions. (3) `Apply` the new vertex value using the output of `gather`. (4) `Scatter` the difference between the old and the new vertex values along the outgoing edges.

To turn a GAS program into a new program, one only needs to add an `extract` phase in the beginning that uses a vertex value v to compute *backward* the value g gathered from the incoming edges of the vertex at the time v was computed. g is then aggregated with a value gathered from the present incoming edges to compute a new value for the vertex. To illustrate, consider the PageRank algorithm that has the following GAS functions:

$$\begin{aligned}
&\text{[GATHER]} && sum \leftarrow \Sigma_{e \in in(v)} e.data \\
&\text{[APPLY]} && v.pr \leftarrow (0.15 + 0.85 * sum)/v.numOutEdges \\
&\text{[SCATTER]} && \forall e \in out(v) : e.data \leftarrow v.pr
\end{aligned}$$

Adding the `extract` phase produces:

$$\begin{aligned}
&\text{[EXTRACT]} && oldsum \leftarrow (v.pr * v.numOutEdges - 0.15)/0.85 \\
&\text{[GATHER]} && newsum \leftarrow oldsum + \Sigma_{e \in in(v)} e.data \\
&\text{[APPLY]} && newpr \leftarrow (0.15 + 0.85 \times newsum)/v.numOutEdges; \\
& && oldpr \leftarrow v.pr; \;\; v.pr \leftarrow newpr \\
&\text{[SCATTER]} && \forall e \in out(v) : e.data \leftarrow newpr - oldpr
\end{aligned}$$

In this example, `extract` reverses the PageRank computation to obtain the old aggregated value *oldsum*, on top of which the new contributions of the present incoming edges are added by `gather`. `Apply` keeps its original semantics and computes a new PageR-

ank value. Before this new value is saved on the vertex, the delta between the old and new is computed and propagated along the outgoing edges in `scatter`.

An alternative way to implement the accumulation-based computation is to save the value gathered from incoming edges on each vertex (*e.g.*, *oldsum*) together with the vertex value so that we do not even need the `extract` phase. However, this approach doubles the size of vertex data which also negatively impacts the time cost due to the extremely large numbers of vertices in real-world graphs. In fact, the `extract` phase does not create extra computation in most cases: after simplification and redundancy elimination, the PageRank formulas using the traditional GAS model and the accumulation-based model require the same amount of computation:

$$pr = \begin{cases} 0.15 + 0.85 \times sum & \dots \text{ traditional} \\ v.pr + 0.85 \times sum & \dots \text{ accumulation-based} \end{cases}$$

Impact on the Delay Buffer

Since the contribution of each incoming edge can be incrementally added onto the vertex value, this model does not need the presence of all incoming edges to compute vertex values. Hence, it significantly decreases the number of vertices whose computation needs to be delayed, reducing the need to frequently run shadow iterations.

If a vertex has a missing outgoing edge, delay is needed. To illustrate, consider again the $u \to v \to w$ example in the beginning of Section 6.2. Since the edge $v \to w$ is missing, although v gets an updated value, the value cannot be pushed out. We have to delay the computation until a shadow iteration in which $v \to w$ is brought into memory.

More precisely, v's `gather` and `apply` can still be executed right away; only its `scatter` operation needs to be delayed, because the target of the `scatter` is unknown due to the missing outgoing edge.

Hence, for each vertex, we execute `gather` and `apply` instantly to obtain the result value r. If the vertex has a missing outgoing edge, the vertex is pushed into the delay buffer together with the value r. Each entry in the buffer now becomes a vertex-value pair. In the next shadow iteration, when this missing edge is brought into memory, r will be pushed through the edge and be propagated.

Since a vertex with missing outgoing edges can be encountered multiple times before a shadow iteration is scheduled, the delay buffer may contain multiple entries for the same vertex, each with a different delta value. Naïvely propagating the most recent increment is incorrect due to the accumulative nature of the model; the consideration of *all* the entries for the vertex is thus required. Hence, we require the developer to provide an additional *aggregation function* that takes as input an ordered list of all delta values for a vertex recorded in the delay buffer and generates the final value that can be propagated to its outgoing edges (details are given in Section 6.3.2).

Although our programming model exposes the `extract` phase to the user, not all algorithms need this phase. For example, algorithms such as ShortestPath and ConnectedComponents can be easily coded in a traditional way, that is, edge data still represent actual values (*i.e.*, paths or component IDs) instead of value changes. This is because in those algorithms, vertex values are in *discrete domains* and `gather` is done by monotonically selecting a value from one incoming edge instead of accumulating values from all incoming

edge values. For instance, ShortestPath and ConnectedComponents use selection functions (min/max) to aggregate contributions of incoming edges.

To make the differences between algorithm implementations transparent to the users, we allow users to develop normal GAS functions without thinking about what data to push along edges. The only additional function the user needs to add is `extract`. Depending on whether `extract` is empty, our system *automatically* determines the meaning of edge data and how it is pushed out.

6.3.2 Model Applicability and Correctness

It is important to understand precisely what algorithms can and cannot be implemented under the accumulation-based model. There are three important questions to ask about applicability: (1) what is the impact of *incremental computation* on graph algorithms, (2) what is the impact of *delay* on those algorithms, and (3) is the computation still correct when vertex updates are delayed?

Impact of Incremental Computation

An algorithm can be correctly implemented under our accumulation-based model if the composition of its `apply` and `gather` is distributive on some aggregation function. More formally, if vertex v has n incoming edges $e_1, e_2, \ldots e_n$, v's computation can be expressed under our accumulation-based model iff there exists an aggregation function[1] f *s.t.*

$$\texttt{apply}(\texttt{gather}(e_1, \ldots, e_n)) = f(\texttt{apply}(\texttt{gather}(e_1)), \ldots, \texttt{apply}(\texttt{gather}(e_n)))$$

Algorithms	**Aggr. Func.** f
Reachability, MaxIndependentSet	or
TriangleCounting, SpMV, PageRank, HeatSimulation, WaveSimulation, NumPaths	sum
WidestPath, Clique	max
ShortestPath, MinmialSpanningTree, BFS, ApproximateDiameter, ConnectedComponents	min
BeliefPropagation	product
BetweennessCentrality, Conductance, NamedEntityRecognition, LDA, ExpectationMaximization, AlternatingLeastSquares	user-defined aggregation function
GraphColoring, CommunityDetection	N/A

Table 6.1: A list of algorithms used as subjects in the following works and their aggregation functions if implemented under our model: GraphChi [73], GraphLab [85], ASPIRE [129], X-Stream [103], GridGraph [153], GraphQ [134], GraphX [45], PowerGraph [44], Galois [92], Ligra [114], Cyclops [22], and Chaos [102].

For most graph algorithms, we can easily find a function f on which their computation is distributive. Table 6.1 shows a list of 24 graph algorithms studied in recent graph processing works and our accumulation-based model works for all but two. For example, one of these two algorithms is GraphColoring, where the color of a vertex is determined by the colors of all its neighbors (coming through its incoming edges). In this case, it is impossible to compute the final color by applying `gather` and `apply` on different neighbors' colors separately and aggregating these results. For the same reason CommunityDetection cannot be correctly expressed as an incremental computation.

Once function f is found, it can be used to aggregate values from multiple entries of the same vertex in the delay buffer, as described earlier in Section 6.3.1. We provide a set of built-in f from which the user can choose, including *and*, *or*, *sum*, *product*, *min*, *max*, *first*,

[1]The commutative & associative properties from `gather` get naturally lifted to aggregation function f.

		Iteration				
	V/E	**0**	**1**	**2**	**3**	**4 (Shadow)**
	u	$[0, I_u]$	$[I_u, I_u]$	$[I_u, a]$	$[a, b]$	$[b, x]$
No Delay	$u \to v$	$[0, I_u]$	$[I_u, 0]$	$[0, a - I_u]$	$[a - I_u, b - a]$	$[b - a, x - b]$
	v	$[0, I_v]$	$[I_v$,AP(EX(I_v)+I_u)]	[AP(EX(I_v)+I_u), AP(EX(I_v)+I_u)]	[AP(EX(I_v)+I_u), AP(EX(I_v)+a)]	[AP(EX(I_v)+a), AP(EX(I_v)+b)]
Delay	$u \to v$	$[0, I_u]$	$[I_u, 0]$	Missing	Missing	$[b - I_u, x - b]$
	v	$[0, I_v]$	$[I_v$, AP(EX(I_v)+I_u)]	[AP(EX(I_v)+I_u), AP(EX(I_v)+I_u)]	[AP(EX(I_v)+I_u), AP(EX(I_v)+I_u)]	[AP(EX(I_v)+I_u), AP(EX(I_v)+b)]

Table 6.2: A comparison between PageRank executions with and without delays under the accumulation-based model; for each vertex and edge, we use a pair [a, b] to report its pre-(a) and post-iteration (b) value. Each vertex u (v) has a value 0 before it receives an initial value I_u (I_v) in Iteration 0; `EX` and `AP` represent function `Extract` and `Apply`, respectively.

and *last*. For instance, PageRank uses *sum* that produces the final delta by summing up all deltas in the buffer, while ShortestPath only needs to compute the minimum of these deltas using *min*. The user can also implement her own for more complicated algorithms that perform numerical computations.

For graph algorithms with non-distributive `gather` and `apply`, using dynamic partitions delays computation for a great number of vertices, making overhead outweigh benefit. In fact, we have implemented GraphColoring in our system and only saw slowdowns in the experiments. Hence, our optimization provides benefit only for distributive algorithms.

Impact of Delay

To understand the impact of delay, we draw a connection between our computation model with the staleness-based (*i.e.*, relaxed consistency) computation model [129, 27]. The staleness-based model allows computation to be performed on stale values but guarantees correctness by ensuring that all updates are visible at some point during processing (by either using refresh or imposing a staleness upper-bound). This is conceptually similar to our computation model with delays: for vertices with missing outgoing edges, their out-neighbors would operate on stale values until the next shadow iteration.

Since a shadow iteration "refreshes" all stale values, the frequency of performing these shadow iterations bounds the maximum staleness of edge values. Hence, any algorithm that can correctly run under the relaxed consistency model can also safely run under our model. Moreover, the frequency of shadow iterations has no impact on the correctness of such algorithms, as long as they do occur and flush the delayed updates. In fact, all the algorithms in Table 6.1 would function correctly under our delay-based model. However, their performance can be degraded if they cannot employ incremental computation.

Delay Correctness Argument

While our delay-based model shares similarity with the staleness-based model, the correctness of a specific algorithm depends on the aggregation function used for the algorithm. Here we provide a correctness argument for the aggregation functions we developed for the five algorithms used in our evaluation: PageRank, BeliefPropagation, HeatSimulation, ConnectedComponents, and MultipleSourceShortestPath; similar arguments can be used for other algorithms in Table 6.1.

We first consider our implementation of PageRank that propagates changes in page rank values along edges. Since BeliefPropagation and HeatSimulation perform similar computations, their correctness can be reasoned in the same manner. For a given edge $u \rightarrow v$, Table 6.2 shows, under the accumulation-based computation, how the values carried by vertices and edges change across iterations with and without delays.

We assume that each vertex u (v) has a value 0 before it is assigned an initial value I_u (I_v) in Iteration 0 and vertex v has only one incoming edge $u \rightarrow v$. At the end of Iteration 0, both vertices have their initial values because the edge does not carry any

value in the beginning. We further assume that in Iteration 1, the value of vertex u does not change. That is, at the end of the iteration, u's value is still I_u and, hence, the edge will not be loaded in Iteration 2 and 3 under the delay-based model.

We compare two scenarios in which delay is and is not enabled and demonstrate that the same value is computed for v in both scenarios. Without delay, the edge value in each iteration always reflects the change in u's values. v's value is determined by the four functions described earlier. For example, since the value carried by the edge at the end of Iteration 0 is I_u, v's value in Iteration 1 is updated to $\texttt{apply}(\texttt{gather}(\texttt{extract}(I_v),\ I_u)))$. As `gather` is `sum` in PageRank, this value reduces to $\texttt{AP}(\texttt{EX}(I_v) + I_u)$. In Iteration 2, the value from the edge is 0 and thus v's value becomes $\texttt{AP}(\texttt{EX}(\texttt{AP}(\texttt{EX}(I_v) + I_u)) + 0)$. Because `EX` is an inverse function of `AP`, this value is thus still $\texttt{AP}(\texttt{EX}(I_v) + I_u)$. Using the same calculation, we can easily see that in Iteration 4 v's value is updated to $\texttt{AP}(\texttt{EX}(I_v) + b)$.

With delay, the edge will be missing in Iteration 2 and 3, and hence, we add two entries $(u,\ a - I_u)$ and $(u,\ b - a)$ into the delay buffer. During the shadow iteration, the edge is loaded back into memory. The aggregation function `sum` is then applied on these two entries, resulting in value $b - I_u$. This value is pushed along $u \rightarrow v$, leading to the computation of the following value for v:

$$\texttt{AP}(\texttt{EX}(\texttt{AP}(\texttt{EX}(I_v) + I_u)) + (b - I_u))$$
$$\Rightarrow \texttt{AP}(\texttt{EX}(I_v) + I_u + b - I_u)$$
$$\Rightarrow \texttt{AP}(\texttt{EX}(I_v) + b)$$

which is the same as the value computed without delay.

This informal correctness argument can be used as the base case for a formal proof by induction on iterations. Although we have one missing edge in this example, the argument can be easily extended to handle multiple missing edges since the `gather` function is associative.

For ShortestPaths and ConnectedComponents, they do not have an `extract` function and their contributions are gathered by the selection function *min*. Since a dynamic shard can never have edges that are not part of its corresponding static shard, vertex values (*e.g.*, representing path and component IDs) in the presence of missing edges are always greater than or equal to their actual values. It is easy to see that the aggregation function *min* ensures that during the shadow iteration the value a of each vertex will be appropriately overridden by the minimum value b of the delayed updates for the vertex if $b \leq a$.

6.3.3 Generalization to Edge-Centricity

Note that the dynamic partitioning techniques presented in this work can be easily applied to edge-centric systems. For example, X-Stream [103] uses an unordered edge list and a scatter-gather computational model, which first streams in the edges to generate updates, and then streams in the generated updates to compute vertex values. To enable dynamic partitioning, dynamic edge lists can be constructed based on the set of changed vertices from the previous iterations. This can be done during the scatter phase by writing to disk the required edges whose vertices are marked dirty.

Hence, later iterations will stream in smaller edge lists that mainly contain the necessary edges. Similarly to processing dynamic shards, computations in the presence of missing edges can be delayed during the gather phase when the upcoming scatter phase

cannot stream in the required edges. These delayed computations can be periodically flushed by processing them during shadow iterations in which the original edge list is made available.

GridGraph [153] is a recent graph system that uses a similar graph representation as used in GraphChi. Hence, our shard-based techniques can be applied directly to partitions in GridGraph. As GridGraph uses very large static partitions (that can accommodate tens of millions of edges), larger performance benefit may be seen if our optimization is added. Dynamic partitions can be generated when edges are streamed in; computation delayed due to missing edges can be detected when vertices are streamed in.

6.4 Optimizing Shard Creation

To maximize net gains, it is important to find a sweet spot between the cost of creating a dynamic shard and the I/O reduction it provides. This section discusses an optimization and analyzes its performance benefit.

6.4.1 Optimization

Creating a dynamic shard at each iteration is an overkill because many newly created dynamic shards provide only small additional reduction in I/O that does not justify the cost of creating them. Therefore, we create a new dynamic shard after several iterations, allowing the creation overhead to be easily offset by the I/O savings.

Furthermore, to maximize edge reuse and reduce delay frequencies, it is useful to include into dynamic shards edges that may be used in *multiple* subsequent iterations. We found that using shadow iterations to create dynamic shards strikes a balance between

Inputs	Type	#Vertices	#Edges	PMSize	#SS
LiveJournal (LJ) [5]	Social Network	4.8M	69M	1.3GB	3
Netflix (NF) [9]	Recomm. System	0.5M	99M	1.6GB	20
UKDoman (UK) [11]	Web Graph	39.5M	1.0B	16.9GB	20
Twitter (TT) [72]	Social Network	41.7M	1.5B	36.3GB	40
Friendster (FT) [40]	Social Network	68.3M	2.6B	71.6GB	80
YahooWeb (YW) [137]	Web Graph	1.4B	6.6B	151.3GB	120

Table 6.3: Input graphs used; PMSize and SS report the peak in-memory size of each graph structure (without edge values) and the number of static shards created in GraphChi, respectively. The in-memory size of a graph is measured as the maximum memory consumption of a graph across the five applications; LJ and NF are relatively small graphs while UK, TT, FT, YW are billion-edge graphs larger than the 8GB memory size; YW is the largest real-world graph publicly available; all graphs have highly skewed power-law degree distributions.

I/O reduction and overhead of delaying computations – new shards are created only during shadow iterations; we treat edges that were updated after the previous shadow iteration as dirty and include them all in the new dynamic shards. The intuition here is that by considering an "iteration window" rather than one single iteration, we can accurately identify edges whose data have truly stabilized, thereby simultaneously reducing I/O and delays.

The first shadow iteration is triggered when the percentage of updated edges p in an iteration drops below a threshold value. The frequency of subsequent shadow iterations depends upon the size of the delay buffer d — when the buffer size exceeds a threshold, a shadow iteration is triggered. Hence, the frequency of shard creation is adaptively determined, in response to the progress towards convergence. We used $p = 30\%$ and $d = 100$KB in our experiments and found them to be effective.

Inputs	#Vertices	#Edges	SizeToMem
LiveJournal (LJ)	4.8M	69M	0.2×
Netflix (NF)	0.5M	99M	0.2×
UKDoman (UK)	39.5M	1.0B	2.4×
Twitter (TT)	41.7M	1.5B	5.2×
Friendster (FT)	68.3M	2.6B	10.2×
YahooWeb (YW)	1.4B	6.6B	21.6×

Table 6.4: Input graphs used; PMSize and SS report the peak in-memory size of each graph structure (without edge values) and the number of static shards created in GraphChi, respectively. The in-memory size of a graph is measured as the maximum memory consumption of a graph across the five applications; LJ and NF are relatively small graphs while UK, TT, FT, YW are billion-edge graphs larger than the 8GB memory size; YW is the largest real-world graph publicly available; all graphs have highly skewed power-law degree distributions.

6.4.2 I/O Analysis

We next provide a rigorous analysis of the I/O costs. We show that the overhead of shard loading in shadow iterations can be easily offset from the I/O savings in regular non-shadow iterations. We analyze the I/O cost in terms of the number of data blocks transferred between disk and memory. Let b be the size of a block in terms of the number of edges and E be the edge set of the input graph. Let AE_i (*i.e.*, active edge set) represent the set of edges in the dynamic shards created for iteration i. Here we analyze the cost of regular iterations and shadow iterations separately for iteration i.

During regular iterations, processing is done using the static shards in the first iteration and most recently created dynamic shards during later iterations. Each edge can be read at most twice (*i.e.*, when its source and target vertices are processed) and written once (*i.e.*, when the value of its source vertex is pushed along the edge). Thus,

$$C_i \leq \begin{cases} \frac{3|E|}{b} & \text{with static shards} \\ \frac{3|AE_i|}{b} & \text{with dynamic shards} \end{cases} \tag{6.1}$$

In a shadow iteration, the static shards and all intermediate dynamic shards are read, the updated edges are written back to static shards, and a new set of dynamic shards are created for the next iteration. Since we only append edges onto existing dynamic shards in regular iterations, there is only one set of dynamic shards between any consecutive shadow iterations. Hence, the I/O cost is:

$$C_i \leq \frac{3|E|}{b} + \frac{2|AE_{LS}|}{b} + \frac{|AE_i|}{b} \tag{6.2}$$

where AE_{LS} is the set of edges in the dynamic shards created by the last shadow iteration. Clearly, C_i is larger than the cost of static shard based processing (*i.e.*, $\frac{3|E|}{b}$).

Eq. 6.1 and Eq. 6.2 provide a useful insight on how the overhead of a shadow iteration can be amortized across regular iterations. Based on Eq. 6.2, the extra I/O cost of a shadow iteration over a regular static-shard-based iteration is $\frac{2|AE_{LS}|}{b} + \frac{|AE_i|}{b}$. Based on Eq. 6.1, the I/O saving achieved by using dynamic shards in a regular iteration is $(\frac{3|E|}{b} - \frac{3|AE_i|}{b})$.

We assume that d shadow iterations have been performed before the current iteration i and hence, the frequency of shadow iterations is $\frac{i}{d}$ (for simplicity, we assume i is multiple of d). This means, shadow iteration occurs once every $\frac{i}{d} - 1$ regular iterations.

In order for the overhead of a shadow iteration to be wiped off by the savings in regular iterations, we need:

$$(\frac{i}{d} - 1) \times (\frac{3|E|}{b} - \frac{3|AE_i|}{b}) \geq \frac{2|AE_{LS}|}{b} + \frac{|AE_i|}{b}$$

After simplification, we need to show:

$$(\frac{i}{d} - 1) \times 3|E| - (\frac{3i}{d} - 2) \times |AE_i| - 2|AE_{LS}| \geq 0 \tag{6.3}$$

Since AE_{LS} is the set of edges in the dynamic shards before Iteration i, we have $|AE_{LS}| \leq |AE_i|$ as we only append edges after that shadow iteration. We thus need to show:

$$\begin{aligned} &(\frac{i}{d} - 1) \times 3|E| - (\frac{3i}{d} - 2) \times |AE_i| - 2|AE_i| \geq 0 \\ \Longrightarrow &\frac{|E|}{|AE_i|} \geq \frac{\frac{i}{d}}{\frac{i}{d} - 1} \end{aligned}$$

The above inequality typically holds for any frequency of shadow iterations $\frac{i}{d} > 1$. For example, if the frequency of shadow iterations $\frac{i}{d}$ is 3, $\frac{|E|}{|AE_i|} \geq 1.5$ means that as long as the total size of static shards is 1.5 times larger than the total size of (any set of) dynamic shards, I/O efficiency can be achieved by our optimization. As shown in Figure 6.3a, after about 10 iterations, the percentage of updated edges in each iteration goes below 15%. Although unnecessary edges are not removed in each iteration, the ratio between $|E|$ and $|AE_i|$ is often much larger than 1.5, which explains the I/O reduction.

6.5 Evaluation

Our evaluation uses five applications including PageRank (PR) [94], MultipleSourceShortestPath (MSSP), BeliefPropagation (BP) [60], ConnectedComponents (CC) [152], and HeatSimulation (HS). They belong to different domains such as social network analysis, machine learning, and scientific simulation. They were implemented using our accumulation-based GAS programming model. Six real-world graphs, shown in Table 6.4, were chosen as inputs for our experiments.

All experiments were conducted on an 8-core commodity Dell machine with 8GB main memory, running Ubuntu 14.04 kernel 3.16, a representative of low-end PCs regular users have access to. Standard Dell 500GB 7.2K RPM HDD and Dell 400GB SSD were used as secondary storage, both of which were connected via SATA 3.0Gb/s interface. File system caches were flushed before running experiments to make different executions comparable.

Two relatively small graphs LJ and NF were chosen to understand the scalability trend of our technique. The other four graphs UK, TT, FT, and YW are larger than memory by 2.4×, 5.2×, 10.2×, and 21.6× respectively.

6.5.1 Overall Performance

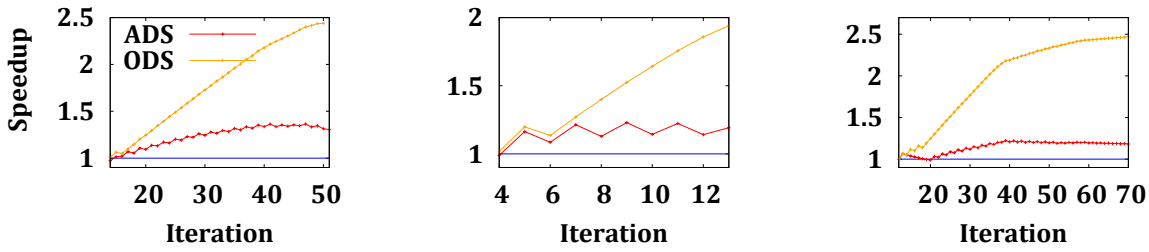
We compared our modified GraphChi extensively with the Baseline (BL) GraphChi that processes static shards in parallel. To provide a better understanding of the impact of the shard creation optimization stated in Section 6.4, we made two modifications, one that creates dynamic shards aggressively (ADS) and a second that uses the optimization in Section 6.4 (ODS). We first report the performance of our algorithms over the first five graphs on HDD in Table 6.5.

G	Version	PR	BP	HS	MSSP	CC
LJ	**BL**	630	639	905	520	291
	ADS	483	426	869	535	296
	ODS	258	383	321	551	263
NF	**BL**	189	876	238	1,799	190
	ADS	174	597	196	1,563	177
	ODS	158	568	164	1,436	178
UK	**BL**	31,616	19,486	21,620	74,566	14,346
	ADS	23,332	15,593	35,200	76,707	14,742
	ODS	14,874	14,227	12,388	67,637	12,814
TT	**BL**	83,676	47,004	75,539	109,010	22,650
	ADS	61,994	38,148	67,522	97,132	21,522
	ODS	47,626	28,434	30,601	84,058	21,589
FT	**BL**	130,928	100,690	159,008	146,518	50,762
	ADS	85,788	84,502	176,767	143,798	50,831
	ODS	87,112	51,905	63,120	127,168	42,956

Table 6.5: A comparison on execution time (seconds) among Baseline (BL), ADS, and ODS.

We ran each program until it converged to evaluate the full impact of our I/O optimization. We observed that for each program the numbers of iterations taken by Baseline and ODS are almost the same. That is, despite the delays needed due to missing edges, the accumulation-based computation and shard creation optimizations minimize the vertices that need to be delayed, yielding the same convergence speed in ODS. ADS can increase the number of iterations in a few cases due to the delayed convergence. On average, ADS and ODS achieve an up to 1.2× and 1.8× speedup over *Baseline*.

PR, BP, and HS are computation-intensive programs and they operate on large working sets. For these three programs, on average ADS speeds up graph processing by 1.53×, 1.50× and 1.22×, respectively. ODS performs much better providing speedups of 2.44×, 1.94×, and 2.82× respectively. The optimized version ODS performs better than the aggressive version ADS because ODS is likely to eliminate edges after the computation



(a) PR on LJ. (b) BP on FT. (c) HS on TT.

Figure 6.6: Speedups achieved per iteration.

of their source vertices becomes stable, and thus edges that will be useful in a few iterations are likely to be preserved in dynamic shards. ODS consistently outperforms the baseline. While ADS outperforms the baseline in most cases, eliminating edges aggressively delays the algorithm convergence for HS on UK (*i.e.*, by 20% more iterations).

MSSP and CC require less computation and they operate on smaller and constantly changing working sets. Small benefits were seen from both ADS (1.15× speedup) and ODS (1.30× speedup), because eliminating edges achieves I/O efficiency at the cost of locality.

Figure 6.6 reports a breakdown of speedups on iterations for PR, BP, and HS. Two major observations can be made here. First, the performance improvement increases as the computation progresses, which confirms our intuition that the amount of useful data decreases as the computation comes close to the convergence. Second, the improvements from ADS exhibit a saw-tooth curve, showing the need of the optimizations in ODS: frequent drops in speedups are due to frequent shard creation and shadow iterations. These time reductions are entirely due to reduced I/O because the numbers of iterations taken by ODS and Baseline are almost always the same.

	PR	BP	HS
BL	153h:33m	80h:19m	147h:48m
ODS	92h:26m	54h:29m	92h:7m
Speedup	**1.66**×	**1.47**×	**1.60**×

Table 6.6: PR, BP and HS on YW.

Since the YW graph is much larger and takes much longer to run, we evaluate ODS for PR, BP and HS whose performance is reported in Table 6.6. ODS achieves a 1.47 – 1.60× speedup over Baseline for PR, BP and HS.

Performance on SSD

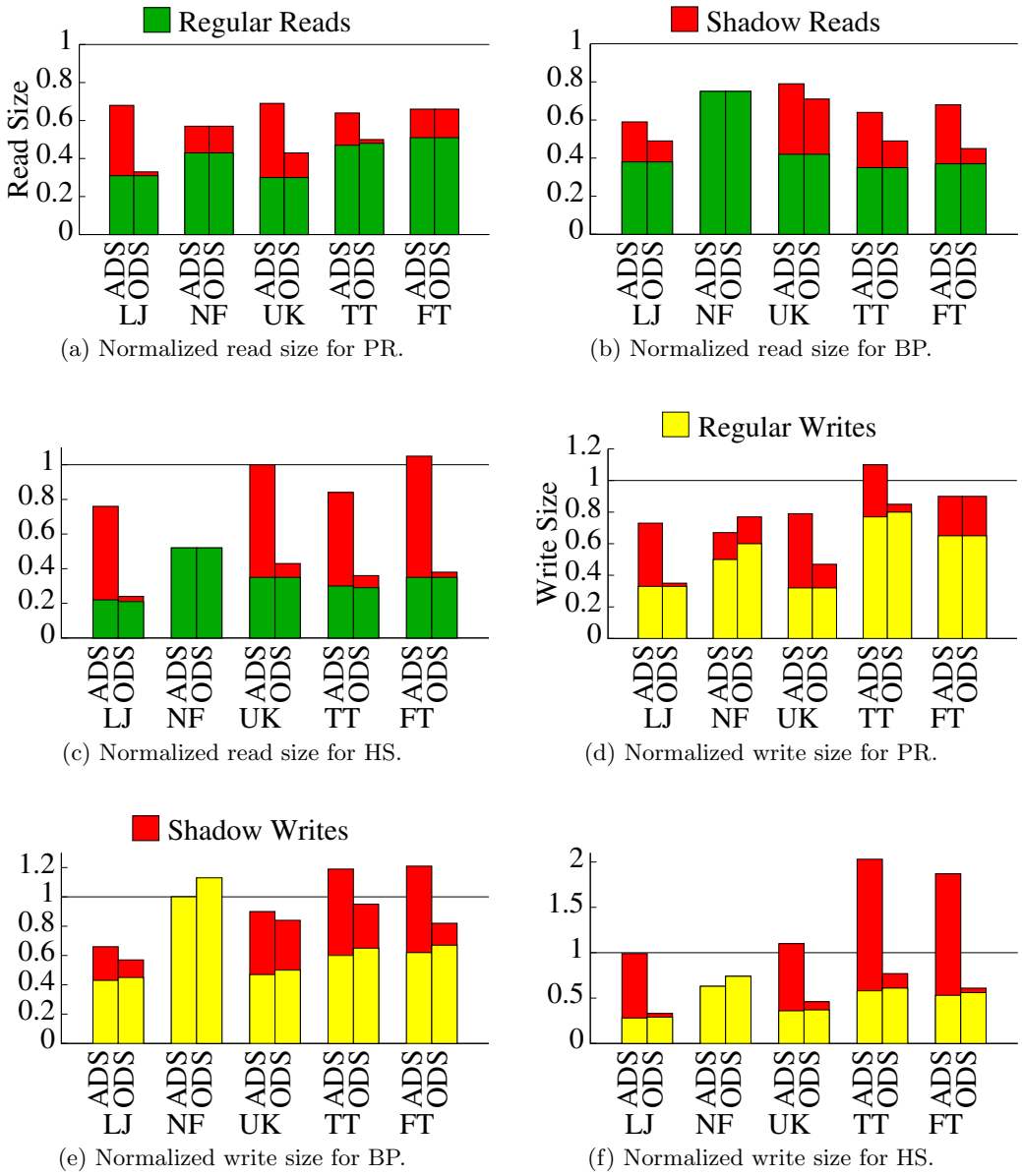
To understand whether the proposed optimization is still effective when high-bandwidth SSD is used, we ran experiments for PR and BP on a machine with the same configuration except that SSD is employed to store shards. We found that the performance benefits are consistent when SSD is employed: on average, ADS accelerates PR, BP and HS by 1.25×, 1.18× and 1.14× respectively, whereas ODS speeds them up by 1.67×, 1.52× and 1.91×.

Note that our techniques are independent of the storage type and the performance benefits are mainly achieved by reducing shard loading time. This roughly explains why a lower benefit is seen on SSD than on HDD – for example, compared to HDD, the loading time for FT on SSD decreases by 8%, 11% and 7% for PR, BP and HS, respectively.

6.5.2 I/O Analysis

Data Read/Written

Figure 6.7 shows the amount of data read and written during the graph processing in the modified GraphChi, normalized *w.r.t. Baseline*. Reads and writes that occur during shadow



(a) Normalized read size for PR.

(b) Normalized read size for BP.

(c) Normalized read size for HS.

(d) Normalized write size for PR.

(e) Normalized write size for BP.

(f) Normalized write size for HS.

Figure 6.7: Read and write size for different benchmarks normalized w.r.t. the baseline.

iterations are termed *shadow reads* and *shadow writes*. No shadow iteration has occurred when some applications were executed on the Netflix graph (*e.g.*, in Figures 6.7 (b), (c), (e), and (f)), because processing converges quickly and dynamic shards created once are able to capture the active set of edges until the end of execution.

Clearly, ODS reads/writes much less data than both Baseline and ADS. Although shadow iterations incur additional I/O, this overhead can be successfully offset from the savings in regular iterations. ADS needs to read and write more data than Baseline in some cases (*e.g.*, Friendster in Figure 6.7c, Twitter in Figure 6.7d and Figure 6.7e). This shows that creating dynamic shards too frequently can negatively impact performance.

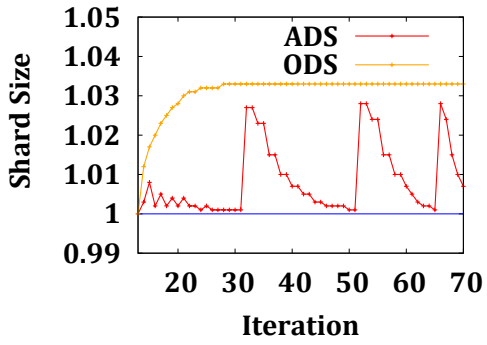
Size of Dynamic Shards

To understand how well ADS and ODS create dynamic shards, we compare the sizes of intermediate dynamic shards created using these two strategies. Figure 6.8 and Figure 6.9 shows the change of the sizes of dynamic shards as the computation progresses, normalized *w.r.t.* the size of an ideal shard. The ideal shard for a given iteration includes only the edges which were updated in the previous iteration, and hence, it contains the minimum set of edges necessary for the next iteration. Note that for both ADS and ODS, their shard sizes are close to the ideal sizes. In most cases, the differences are within 10%.

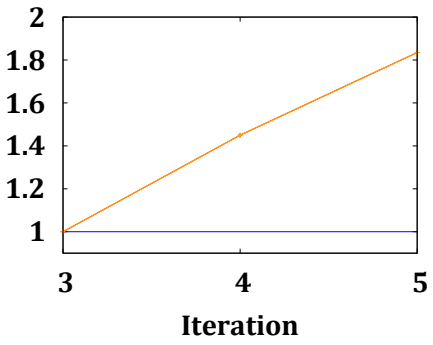
It is also expected that shards created by ODS are often larger than those created by ADS. Note that patterns exist in shard size changes for ADS such as HS on LJ (Figure 6.8a) and FT (Figure 6.8e). This is because the processing of delayed operations (in shadow iterations) over high-degree vertices causes many edges to become active and be included in new dynamic shards.

Edge Utilization

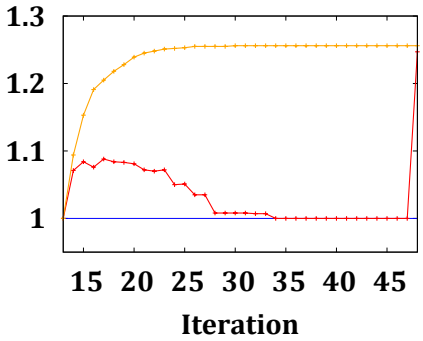
Figure 6.10 reports the average *edge utilization rates* (EUR) for ADS and ODS, and compares them with that of Baseline. The average edge utilization rate is defined as the per-



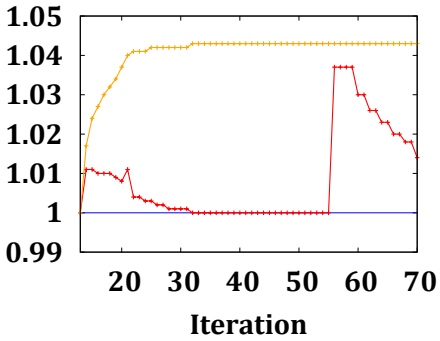
(a) HS on LJ.



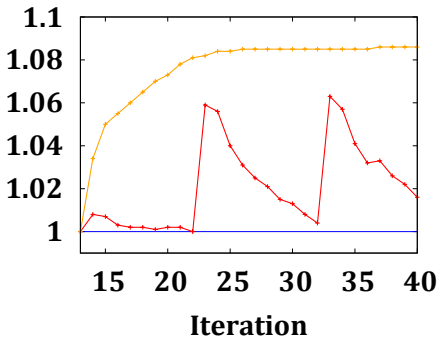
(b) HS on NF.



(c) HS on UK.

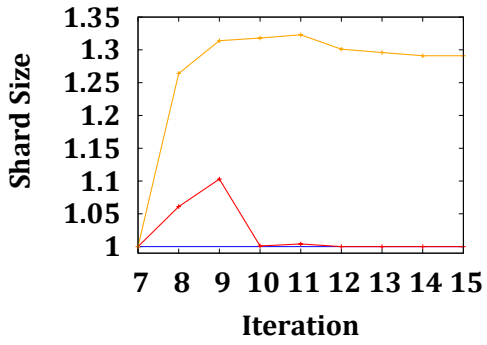


(d) HS on TT.

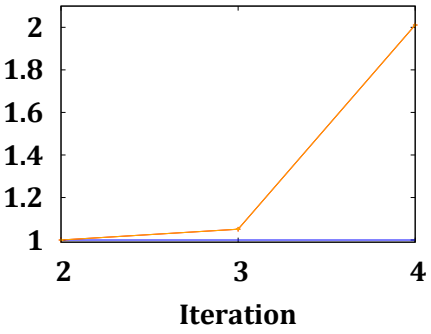


(e) HS on FT.

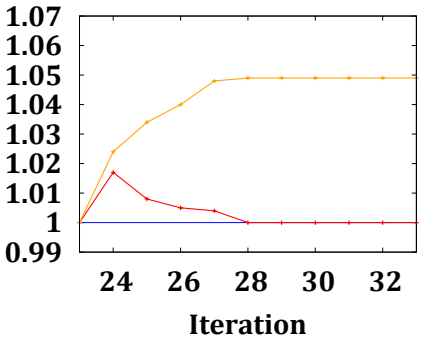
Figure 6.8: The dynamic shard sizes for HS normalized *w.r.t.* the ideal shard sizes as the algorithm progresses.



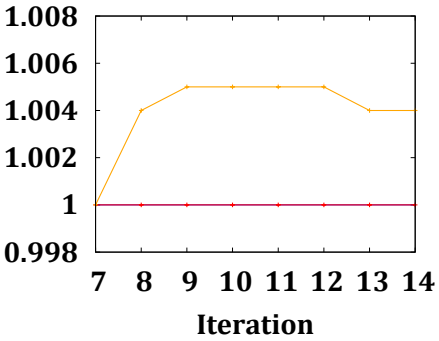
(a) MSSP on LJ.



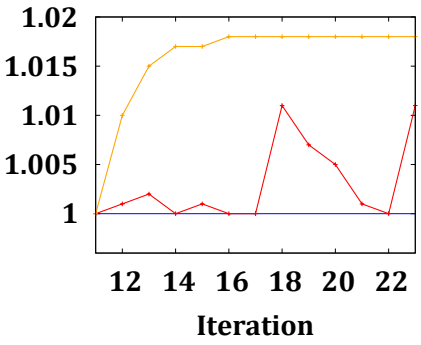
(b) MSSP on NF.



(c) MSSP on UK.



(d) MSSP on TT.



(e) MSSP on FT.

Figure 6.9: The dynamic shard sizes for MSSP normalized *w.r.t.* the ideal shard sizes as the algorithm progresses.

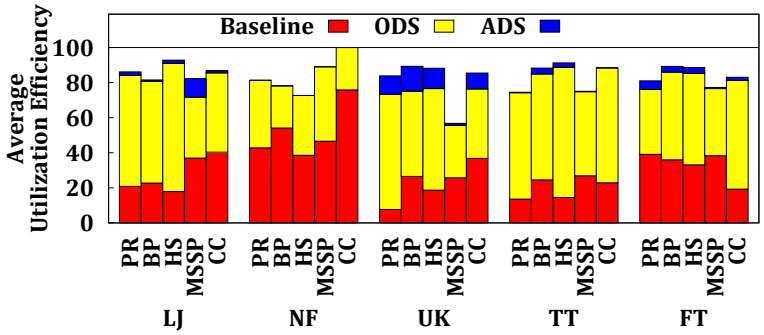


Figure 6.10: Edge utilization rates.

centage of updated edges in a dynamic shard, averaged across iterations. Using dynamic shards highly improves the edge utilization: the EURs for ADS and ODS are between 55% and 92%. For CC on NF, the utilization rate is 100% even for ODS, because computation converges quickly and dynamic shards are created only once. Clearly, ADS has higher EURs than ODS because of its aggressive shard creation strategy. Using static shards throughout the execution leads to a very low EUR for Baseline.

Disk Space Consumption

Figure 6.11 reports the maximum disk space needed to process dynamic shards normalized *w.r.t.* that needed by Baseline. Since we create and use dynamic shards only after vertex computations start stabilizing, the actual disk space it requires is very close to (but higher than) that required by Baseline. This can be seen in Figure 6.11 where the disk consumption increases by 2-28%. Note that the maximum disk space needed is similar for ADS and ODS, because dynamic shards created for the first time take most space; subsequent shards are either smaller (for ADS), or additionally include a small set of active edges (for ODS), which is insignificant to affect the ratio.

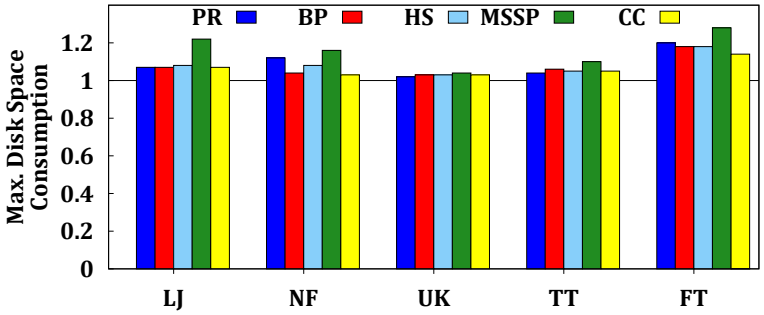


Figure 6.11: Max disk space used.

Delay Buffer Size

With the help of the accumulation-based computation, the delay buffer often stays small throughout the execution. Its size is typically less than few 100KBs. The peak consumption was seen when ConnectedComponent was run on the Friendster graph, and the buffer size was 1.5MB.

6.5.3 Comparisons with X-Stream

Figure 6.12 compares the speedups and the per-iteration savings achieved by ODS and X-Stream over Baseline when running PR on large graphs. The saving per iteration

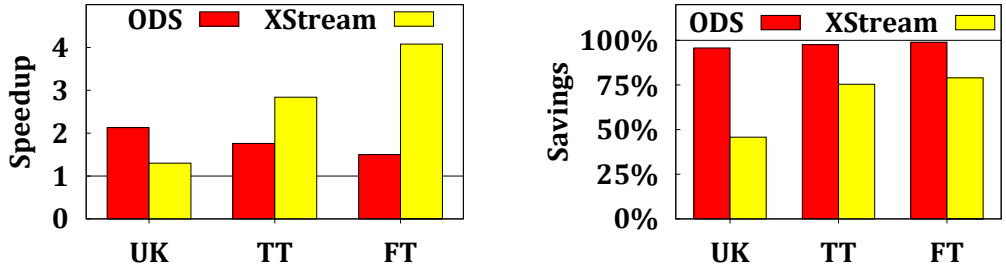


Figure 6.12: Speedups achieved (left) and per-iteration savings in execution time achieved (right) by ODS and X-Stream over Baseline using PR.

was obtained by (1) calculating, for each iteration in which dynamic shards are created, $\frac{Baseline-ODS}{Baseline}$, and (2) taking an average across savings in all such iterations. While the per-iteration savings achieved by dynamic shards are higher than those by X-Stream, ODS is overall slower than X-Stream (*i.e.*, ODS outperforms X-Stream on UK but underperforms it on other graphs).

This is largely expected due to the fundamentally different designs of the vertex- and edge-centric computation models. Our optimization is implemented in GraphChi, which is designed to scan the whole graph multiple times during each iteration, while X-Stream streams edges in and thus only needs one single scan. Hence, although our optimization reduces much of GraphChi's loading time, this reduction is not big enough to offset the time spent on extra graph scans. Furthermore, in order to avoid capturing a large and frequently changing edge set (as described in Section 6.4.1), our optimization for creating and using dynamic shards gets activated after a certain number of iterations (*e.g.*, 20 and 14 for TT and FT, respectively), and these (beginning) iterations do not get optimized.

Although X-Stream has better performance, the proposed optimization is still useful in practice for two main reasons. First, there are many vertex-centric systems being actively used. Our results show that the use of dynamic shards in GraphChi has significantly reduced the performance gap between edge-centricity and vertex-centricity (from 2.74× to 1.65×). Second, our performance gains are achieved only by avoiding the loading of edges that do not carry updated values and this type of inefficiency also exists in edge-centric systems. Speedups should be expected when future work optimizes edge-centric systems using mechanisms proposed in Section 6.3.3.

6.6 Summary

In this chapter, we demonstrated the efficacy of asynchrony for improving out-of-core graph processing by developing dynamic partitions that changes the layout of the partition structure to reduce disk I/O. We leveraged the computation reordering technique to develop a delay based processing model along with accumulative computation that fully utilizes dynamic partitions. Our experiments with GraphChi demonstrated that this optimization has significantly shortened its I/O time and improved its overall performance.

While we chose the out-of-core setting to showcase the effectiveness of leveraging the asynchronous model beyond the distributed processing environment, similar techniques can be developed for various other processing environments like ones including GPUs, FPGAs, non-volatile memories, etc.

Chapter 7

Related Work

This chapter discusses various research works in the literature that address similar issues. We first discuss various works that focus on graph processing and then briefly discuss general purpose solutions based on the relaxed consistency philosophy.

7.1 Graph Processing Solutions

We first discuss various techniques developed to process static graphs, and then present solutions to process dynamic graphs.

7.1.1 Static Graph Processing

There have been many advances in developing frameworks for distributed graph processing. Google's *Pregel* [86] provides a synchronous vertex centric framework for large scale graph processing which uses message passing instead of a shared memory abstraction. *GraphLab* [84] provides a framework for asynchronous execution of machine learning and

data mining algorithms on graphs. It allows users to choose among three different data consistency constraints to balance program correctness and performance. *PowerGraph* [44] provides efficient distributed graph placement and computation by exploiting the structure of power-law graphs. It provides both, synchronous and asynchronous execution and enforces serializability by avoiding adjacent vertex programs from running concurrently. *Pregelix* [12] is a distributed graph processing system based on an iterative dataflow design to handle both in-memory and out-of-core workloads. *GraphX* [45] is a graph processing framework built using the *Apache Spark* [143] distributed dataflow system. [123] presents a graph centric programming model which exposes the partition structure to accelerate convergence by bypassing intermediate messages. Cyclops [22] provides a distributed immutable view, granting vertices read-only accesses to their neighbors and allowing unidirectional communication from master vertices to their replicas. GoFFish [115] presents a sub-graph centric framework that enables programming flexibility while providing communication efficiency. Arabesque [122] focuses on graph mining by providing efficient sub-graph exploration using filter-process computational model based on problem-specific exploration criteria. Chaos [102] utilizes disk space on multiple machines to scale graph processing.

Ligra [114] presents a simple shared memory abstraction for vertex algorithms which is particularly good for problems similar to graph traversal. [92] presents a shared-memory based implementations of these DSLs on a generalized *Galois* [69] system and compares its performance with the original implementations. These frameworks are based on the Bulk Synchronous Parallel (BSP) [126] model and the MapReduce [30] philosophy which allow users to write code from a local perspective and let the system translate

the computations to larger datasets. However, they do not provide support for programming asynchronous algorithms by using stale values (Chapter 2). The ideas we presented throughout this thesis can be incorporated in above frameworks to support asynchronous algorithms. *GRACE* [132], a shared memory based graph processing system, uses message passing and provides asynchronous execution by using stale messages. Since shared-memory processing does not suffer from communication latencies, these systems can perform well for graphs which can fit on a single multicore server. [71] develops techniques to efficiently transform the input graph into a smaller graph and presents a two-phase processing model to leverage incremental computation by using values computed on the transformed graphs while processing the original input graph.

GraphChi [73] provides efficient disk-based graph processing on a single machine for input graphs that cannot fit in memory. As mentioned in Chapter 6, shards are created during pre-processing and are never changed during graph computation, resulting in wasteful I/O. Our work exploits dynamic shards whose data can be dynamically adjustable to reduce I/O. Efforts have been made to reduce I/O using semi-external memory and SSDs. Bishard Parallel Processor [91] aims to reduce non-sequential I/O by using separate shards to contain incoming and outgoing edges. This requires replication of all edges in the graph, leading to disk space blowup. X-Stream [103] uses an edge-centric approach in order to minimize random disk accesses. In every iteration, it streams and processes the entire unordered list of edges during the `scatter` phase and applies updates to vertices in the `gather` phase. Using our approach, *dynamic edge-lists* can be created to reduce wasteful I/O in the `scatter` phase of X-Stream. GridGraph [153] uses partitioned vertex chunks

and edge blocks as well as a dual sliding window algorithm to process graphs residing on disks. It enables selective scheduling by eliminating processing of edge blocks for which vertices in the corresponding chunks are not scheduled. However, the two-level partitioning is still done statically. Conceptually, making partitions dynamic would provide additional benefit over the 2-level partitioning. FlashGraph [150] is a semi-external memory graph engine that stores vertex states in memory and edge-lists on SSDs. It is built based on the assumption that all vertices can be held in memory and a high-speed user-space file system for SSD arrays is available to merge I/O requests to page requests. TurboGraph [50] is an out-of-core computation engine for graph database to process graphs using SSDs. Since TurboGraph uses an adjacency list based representation, algorithms need to be expressed as sparse matrix-vector multiplication, which has a limited applicability because certain algorithms such as triangle counting cannot be expressed in this manner. Graspan [133] is a disk-based graph system that processes program graphs with constant edge additions.

7.1.2 Evolving Graph Processing

This work, similar to our work [127], considers the scenario where an evolving graph is represented as a sequence of snapshots that are then analyzed. Chronos [49] is a storage and execution engine to process temporal graphs. It employs a graph representation that places vertex data from different snapshots together. It exploits the cache locality created by the above graph representation by interleaved (parallel) processing of snapshots in a single- and multi-threaded environment leading to improved L1 data cache and last level cache performance. Both Chronos and FA (Chapter 4) exploit synergy between different graph snapshots by collocating vertex values, *in memory* versus *in messages* respectively.

In presence of FA, Chronos performs *incremental processing* by feeding stable values from the most recent snapshot in the previous batch to all the snapshots in the next batch. This requires Chronos to wait for all the snapshots in the previous batch to finish before the next batch can begin execution. In comparison, our PA (Chapter 4) allows faster processing by feeding potentially unstable (partially computed) values which are available from the most recent snapshot. In addition, in our work we also consider the interplay between FA, PA, and caching in distributed and shared-memory environments. GraphInc [14] is a system which allows incremental processing of graphs by employing memoization of incoming messages to remove redundant vertex computations for same sets of messages.

GraphScope [119] focuses on the encoding of time-evolving graphs to efficiently perform community discovery and change detection. Kan et al. [59] present indexing techniques for detecting simple spatio-temporal patterns in evolving graphs – unlike our work these queries are simple and do not involve iterative algorithms. TEG [38] focuses on partitioning time evolving graphs across nodes of a cluster for distributed processing and implements reachability and subgraph queries. In [65], a distributed graph database system is developed to manage historical data for evolving graphs, supporting temporal queries and analysis. Many *algorithmic approaches* are being developed for improving efficiency. Ren et al. [98] propose *Find-Verify-Fix* approach where from a sequence of snapshots a representative snapshot is constructed. To process a query, first representative snapshot is used and if there is success, the real snapshots are queried to verify and fix the response. Like our work, the Find-Verify-Fix approach also recognizes the synergy between analysis performed on different snapshots. However, this approach is effective when only the struc-

tural properties of graphs are being considered. Desikan et al. [32] propose an incremental PageRank algorithm for evolving graphs that partitions graph into a part whose PageRank values will remain unchanged and the remaining subgraph. This algorithm exploits the underlying principle of first order markov model on which PageRank is based.

7.1.3 Streaming Graph Processing

Custom solutions develop specialized streaming algorithms to solve different problems. They incorporate correctness in the algorithm design either by relaxing the problem constraints or by dealing with edge mutations in specific ways. STINGER [35] uses a novel data structure which enables quick insertions and deletions while allowing parallel traversal of vertices and edges. [34] develops an approximation method for maintaining clustering coefficients using bloom filters. [33, 99] incorporate techniques to correctly maintain connected components information using STINGER by using set intersection of neighborhood vertices to quickly determine connectivity and construct a spanning tree for each component to check for reachability up to root. While checking reachability is expensive, the algorithm relies on multiple concurrent graph traversals to maximize parallelism. In comparison, our trimming solution (Chapter 5) in KickStarter [128] does not need expensive traversals since it relies on level checking. Other custom solutions for connectivity checks upon deletions are: [113] relies on two searches to find component splits whereas [100, 54] maintain graph snapshots for each iteration and use LCA and coloring technique. [141] presents a novel clustering algorithm which is aware of the evolution whereas Fennel [125] proposes a novel partitioning algorithm. [118] proposes greedy, chunking and balancing based heuristics to partition streaming graphs.

Generalized Streaming Graph Processing systems allow users to express graph algorithms. When a query arrives in Tornado [112], it takes the current graph snapshot and branches the execution to a separate loop to compute results using incremental processing. Kineograph [24] is a distributed streaming graph processing system which enables graph mining over fast-changing graphs and uses incremental computation along with push and pull models. [120] proposes the GIM-V incremental graph processing model based upon matrix-vector operations. [98] constructs representative snapshots which are initially used for querying and upon success uses real snapshots. Naiad [90] incorporates differential data flow to perform iterative and incremental algorithms.

Various generalized data stream processing systems [124, 144, 48, 7, 1, 145, 97, 4] have been developed that operate on unbounded structured and unstructured streams which allow window operations, incremental aggregation and instant querying to retrieve timely results. They allow users to develop their own streaming algorithms; note that users must ensure correctness of algorithms. [104] identifies errors in data-stream processing and improves the accuracy of sketch-based algorithms like Count-Min, Frequency-Aware Counting, etc.

7.2 Weak Memory Models

Extensive research has been conducted on weak memory models that relax consistency. A hierarchy of these models can be found in [89] and [53]. We have already discussed the relevant ones; here we provide a complete characterization of the models.

A number of models are too strong for asynchronous algorithms. *Release consistency* [42] and its variants like *lazy release consistency* [63, 140] relax consistency by delaying visibility of updates until certain *specially labeled* accesses. *Causal memory* [2] is weaker than Lamport's *sequential consistency* [75] which guarantees that processes agree on the relative ordering of operations that are *potentially causally related* [74]. *Entry consistency* [10] guarantees consistency only when a thread enters a critical section defined by synchronization variables. *Scope consistency* [57] enforces that all updates in the previous consistency session are visible at the start of current consistency session for the same scope. Even though entry consistency and scope consistency can be used to mimic relaxed coherence, by manually controlling synchronization variables and consistency scopes, none of these models inherently relax the consistency.

Other models are too weak and hence not a good fit for asynchronous algorithms. *Pipelined RAM (PRAM)* [106] provides fast data access similar to our proposed model: on a read, it simply returns the local copy and on write, local values are updated and the new value is broadcast to other processors. However, it allows inconsistent data-views because relative order of updates from different processes can vary. Also, it enforces a strict constraint that all processors must agree on the order of all observed writes by a single processor. This means, broadcast of these updates cannot be skipped, and hence, as shown in [51], the time taken for flow of updates in PRAM increases rapidly as the number of processes increase. This increase in flow of updates can delay the convergence of asynchronous algorithms. Our pull-based model (Chapter 2) allows object values to be skipped since it does not constrain the order of observed writes to different objects and hence, is

more flexible to provide faster convergence. *Slow memory* [56] makes the consistency model weak enough for a read to return some previously written value. This allows the cache to be non-coherent, but requires programming effort to guarantee convergence.

Mermera [52] tries to solve the same problem for iterative asynchronous algorithms by using slow memory. Programs written on mermera handle the correctness and convergence issue by explicitly maintaining a mix of slow writes, coherent writes, and specialized barriers (to flush slow writes). Also, slow memory is based on delayed updates; if it is implemented to support a DSM which is not update based, once the stream of delayed updates enters the memory, local copies will be invalidated and the same issue (waiting for remote fetch) arises. To enable ease of programming and allow intuitive reasoning about execution, our consistency model (Chapter 2) guarantees the progressive reads semantics [129, 130] while still allowing relaxation of consistency for use of stale objects.

Finally, a number of models support bounded staleness. This is same as *delta coherence* as used in *InterWeave* [21], that allows use of objects that are no more than x versions out-of-date. Even though this mechanism proved to be useful to reduce network usage, maintaining a static staleness upper bound x is not useful; a low value of x will only hide few remote fetches because stale objects will quickly become useless while a high value of x can significantly delay the convergence as updates are slowly propagated through the system, allowing many wasteful computations. This issue is also faced by the *stale synchronous parallel* (SSP) model [26]. SSP defines staleness as the number of iterations since the object at hand received its value. Their experiments show that the convergence behavior begins to degrade when the staleness bound is increased past a certain value.

Hence, statically bounding staleness is not the correct approach to improve performance of asynchronous iterative algorithms. The challenge is to allow use of stale objects when up-to-date values are not available but, at the same time, minimize the staleness of such objects. *Delta consistency* introduced in [116] has a similar approach as delta coherence, but enforces a temporal bound on staleness. This requires mechanisms for temporal synchronization to compute the *global virtual time* (GVT). Again, since there is no global ordering for writes from different processors to the same location, correctness semantics need to be externally ensured. Also, none of these models proactively try to maintain low staleness by fetching and updating values. This means, fetches on critical paths are blocked often because the values become too stale which limits their performance benefits.

DSM Coherence Frameworks

Many coherence frameworks, for page as well as object based systems, support multiple coherence schemes to effectively deal with variety of behaviors. *Shasta* [107] is a page based DSM that provides flexibility of varying coherence granularity for shared data structures. *CASHMERe* [67] provides a scalable shared memory which uses page sized coherence blocks. It uses an asynchronous protocol but, the focus is not towards relaxation of coherence. *TreadMarks* [64] is designed to reduce communication for maintaining memory consistency. It uses lazy release consistency and multiple writer based protocols that provide strict consistency guarantee and incurs less communication. In [3] dynamic adaption between single writer and multiple writer protocols is proposed to balance false sharing with computation and memory costs. This work is tangential to our goal which is improving performance of asynchronous iterative algorithms by allowing controlled use of stale objects.

In Chapter 2, we deal at a higher abstraction level by using object based DSM like *Orca* [6] and *Munin* [15]. Munin uses coherence mechanisms based upon object types. Also, relaxation of coherence in Munin is limited in between synchronization points and at them the delayed updates are flushed to remote copies. *Object View* [82] shares similar goals as Munin and Orca; it provides extensions to Java to specify intended use of objects by computation threads. This allows runtime to use low-overhead caching protocols customized to application requirements. *Problem Oriented Object Memory* [68] allows relaxation of strict consistency by letting objects to fall in different consistency models. Since we aim to specifically improve performance of asynchronous algorithms, we do not distinguish objects based on usage types. *Cachet* [110] dynamically adapts across multiple micro-protocols that are optimized based upon access patterns. [16] focuses on reducing communication required to maintain consistency among distributed memories. [151, 58, 108] and others try to relax consistency using basic memory models previously described. Since they inherently aim to provide a consistent DSM, relaxation of consistency is not explored in these works.

Chapter 8

Conclusions and Future Work

8.1 Contributions

In this thesis, we study and leverage the algorithmic asynchrony to improve large-scale graph processing. We first specified the asynchronous processing model in a distributed setting by identifying key properties based on read-write dependences and order of reads to expose the set of legal executions for asynchronous programs. This allowed us to capture the algorithmic intricacies and execution semantics, enabling us to improve asynchronous processing and making it easier to reason about asynchronous execution semantics while leveraging from its benefits. And then, we developed key techniques to exploit the availability of multiple legal executions by choosing faster executions to accelerate the overall processing via reduction in both, communication and computation while processing static and dynamic graphs.

Static Graph Processing

We presented an effective solution for exploiting the asynchronous nature of iterative algorithms for tolerating communication latency in a cluster. We designed a relaxed consistency model and the RCP protocol that allows threads to utilize stale values, and incorporates a policy for refreshing stale values. Together, these features allow an asynchronous algorithm to tolerate communication latency without adversely impacting algorithm's convergence. We studied the semantics of asynchronous distributed processing that enable fault tolerance at reduced costs. We developed confined recovery strategy upon machine failures by constructing alternate PR-Consistent state without discarding any useful work performed on non-failing machines. CoRAL uses locally consistent snapshots that are captured at reduced peak network bandwidth usage for transferring snapshots to remote machines.

Dynamic Graph Processing

We leveraged the asynchrony to accelerate evolving graph processing using two optimizations. Fetch Amortization reduces remote fetches by aggregating similar messages that get exposed due to computation reordering. Processing Amortization accelerates termination of iterative algorithms by carefully using incremental computations. We also developed an efficient runtime technique to process streaming graphs. We achieved this by exploiting the algorithmic asynchrony to develop KickStarter, a dynamic dependence based incremental processing technique that efficiently computes query results while simultaneously providing 100% correctness guarantees in presence of edge deletions.

Out-of-core Processing

Finally, we demonstrated the efficacy of asynchrony across execution environments beyond a distributed setting. In particular, we improved out-of-core graph processing by developing dynamic partitions that changes the layout of the partition structure to reduce disk based I/O. We leveraged the computation reordering technique to develop a delay based processing model along with accumulative computation that fully utilizes dynamic partitions.

8.2 Future Work

While this thesis demonstrated the effectiveness of leveraging asynchrony by modifying the execution semantics to accelerate performance, algorithmic asynchrony can also be exploited by modifying aspects of the data graph. Furthermore, such asynchrony can be exploited across various other domains which demand computations beyond traditional vertex/edge-centric algorithms.

8.2.1 Graph Transformation

While acceleration via graph reduction has been explored in [71], the work requires multiple processing phases to compute the final results because the transformations do not maintain the structural details of the graph. However, novel transformations can be developed such that the transformed graph maintains the structural details to a level that needs minimal processing phases, while still producing accurate results. Taking a step further, transformations can be developed such that synchronous graph algorithms which strictly limit the processing semantics can also leverage from this technique while providing the same processing guarantees.

8.2.2 Graph Partitioning

To process the graph in a distributed environment, it is first partitioned across different nodes in the cluster and each node becomes responsible to process the subgraph partitioned to that node. Various graph partitioning strategies have been explored [84, 44, 129, 23] which minimize the overall communication while balancing computation across all the nodes. Algorithmic aware partitioning strategies can be developed that relax the traditional communication and load balancing constraints, but expose the algorithmic asynchrony in form of structural properties of the partitioned subgraphs. Furthermore, such partitioning strategies can be used to further develop novel processing models that can leverage the exposed properties of the partitioned subgraphs.

8.2.3 Other Graph Applications

Graphs, being ubiquitous, require processing across various domains like software debugging [148, 39] and privacy across networks [41]. These domains introduce new properties and constraints over both, the graph being processed and the algorithms that process the graphs. For example, software control flow graphs are structurally different from social network graphs, exposing an input characteristic that can be exploited for processing. Similarly, various bug detection and privacy algorithms like graph de-anonymization examine graph sub-structures that are usually larger than the bounded-neighborhood for a given vertex or edge, making them difficult to express and parallelize. Since limited study has been performed to generalize these kinds of graph algorithms, characterizing the read-write dependences can expose various relaxable computations making room for asynchrony to be introduced and exploited.

Bibliography

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the borealis stream processing engine. In *CIDR*, volume 5, pages 277–289, 2005.

[2] Mustaque Ahamad, Gil Neiger, James E Burns, Prince Kohli, and Phillip W Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[3] Cristiana Amza, Alan L Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Software dsm protocols that adapt between single writer and multiple writer. In *High-Performance Computer Architecture, 1997., Third International Symposium on*, pages 261–271. IEEE, 1997.

[4] Rajagopal Ananthanarayanan, Venkatesh Basker, Sumit Das, Ashish Gupta, Haifeng Jiang, Tianhao Qiu, Alexey Reznichenko, Deomid Ryabkov, Manpreet Singh, and Shivakumar Venkataraman. Photon: Fault-tolerant and scalable joining of continuous data streams. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 577–588, New York, NY, USA, 2013. ACM.

[5] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group formation in large social networks: Membership, growth, and evolution. In *KDD*, pages 44–54, 2006.

[6] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: a language for parallel programming of distributed systems. *Software Engineering, IEEE Transactions on*, 18(3):190–205, 1992.

[7] Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Eddie Galvez, Jon Salz, Michael Stonebraker, Nesime Tatbul, et al. Retrospective on aurora. *The VLDB Journal*, 13(4):370–383, 2004.

[8] Gérard M Baudet. Asynchronous iterative methods for multiprocessors. *Journal of the ACM (JACM)*, 25(2):226–244, 1978.

[9] James Bennett and Stan Lanning. The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35, 2007.

[10] Brian N Bershad, Matthew J Zekauskas, and J Midway. Shared memory parallel programming with entry consistency for distributed memory multiprocessors. 1991.

[11] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *WWW*, pages 595–601, 2004.

[12] Yingyi Bu. Pregelix: Dataflow-based big graph analytics. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 54:1–54:2, New York, NY, USA, 2013. ACM.

[13] Michael G. Burke and Barbara G. Ryder. A critical analysis of incremental iterative data flow analysis algorithms. *IEEE Trans. Softw. Eng.*, 16(7):723–728, July 1990.

[14] Zhuhua Cai, Dionysios Logothetis, and Georgos Siganos. Facilitating real-time graph mining. In *CloudDB*, pages 1–8, 2012.

[15] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of munin. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, SOSP '91, pages 152–164, New York, NY, USA, 1991. ACM.

[16] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared-memory systems. *ACM Trans. Comput. Syst.*, 13(3):205–243, August 1995.

[17] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and P Krishna Gummadi. Measuring user influence in twitter: The million follower fallacy. *ICWSM*, 10(10-17):30, 2010.

[18] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-based cache coherence in large-scale multiprocessors. *Computer*, 23(6):49–58, 1990.

[19] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[20] Daniel Chazan and Willard Miranker. Chaotic relaxation. *Linear algebra and its applications*, 2(2):199–222, 1969.

[21] DeQing Chen, Chunqiang Tang, Brandon Sanders, Sandhya Dwarkadas, and Michael L. Scott. Exploiting high-level coherence information to optimize distributed shared state. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 131–142, New York, NY, USA, 2003. ACM.

[22] Rong Chen, Xin Ding, Peng Wang, Haibo Chen, Binyu Zang, and Haibing Guan. Computation and communication efficient graph processing with distributed immutable view. In *HPDC*, pages 215–226, 2014.

[23] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *EuroSys*, pages 1:1–1:15, 2015.

[24] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong 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.

[25] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: graph processing at facebook-scale. In *Proc. VLDB Endowment*, 2015.

[26] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R Ganger, Garth Gibson, Kimberly Keeton, and Eric P Xing. Solving the straggler problem with bounded staleness. In *HotOS*, pages 22–22, 2013.

[27] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Exploiting bounded staleness to speed up big data analytics. In *USENIX ATC*, pages 37–48, 2014.

[28] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[29] Manlio De Domenico, Antonio Lima, Paul Mougel, and Mirco Musolesi. The anatomy of a scientific rumor. *arXiv preprint arXiv:1301.2952*, 2013.

[30] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[31] Delicious user-url network dataset, 2014.

[32] Prasanna Desikan, Nishith Pathak, Jaideep Srivastava, and Vipin Kumar. Incremental page rank computation on evolving graphs. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, WWW '05, pages 1094–1095, 2005.

[33] D. Ediger, J. Riedy, D.A. Bader, and H. Meyerhenke. Tracking structure of streaming social networks. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1691–1699, May 2011.

[34] David Ediger, Karl Jiang, Jason Riedy, and David A. Bader. Massive streaming data analytics: A case study with clustering coefficients, 2010.

[35] David Ediger, Rob Mccoll, Jason Riedy, and David A. Bader. Stinger: High performance data structure for streaming graphs, 2012.

[36] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, September 2002.

[37] Ayman Farahat, Thomas LoFaro, Joel C. Miller, Gregory Rae, and Lesley A. Ward. Authority rankings from hits, pagerank, and salsa: Existence, uniqueness, and effect of initialization. *SIAM J. Sci. Comput.*, 27(4):1181–1201, November 2005.

[38] Arash Fard, Amir Abdolrashidi, Lakshmish Ramaswamy, and John A. Miller. Towards efficient query processing on massive time-evolving graphs, 2012.

[39] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 480–491. ACM, 2016.

[40] Friendster network dataset, 2015.

[41] Hao Fu, Aston Zhang, and Xing Xie. Effective social graph deanonymization based on graph structure and descriptive information. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 6(4):49, 2015.

[42] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. *Memory consistency and event ordering in scalable shared-memory multiprocessors*. ACM, 1990.

[43] Vicenç Gómez, Andreas Kaltenbrunner, and Vicente López. Statistical analysis of the social network and discussion threads in slashdot. In *WWW*, pages 645–654.

[44] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[45] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.

[46] James R Goodman. *Cache consistency and sequential consistency*. University of Wisconsin-Madison, Computer Sciences Department, 1991.

[47] Graphlab create. https://dato.com/products/create/, 2016.

[48] STREAM Group et al. Stream: The stanford stream data manager. *IEEE Data Engineering Bulletin, http://www-db. stanford. edu/stream*, 2(003), 2003.

[49] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: A graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 1:1–1:14, New York, NY, USA, 2014. ACM.

[50] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC. In *KDD*, pages 77–85, 2013.

[51] Abdelsalam Heddaya and Himanshu Sinha. An implementation of mermera: a shared memory system that mixes coherence with non-coherence. Technical report, Boston University Computer Science Department, 1993.

[52] Abdelsalam Heddaya and Himanshu Sinha. An overview of mermera: A system and formalism for non-coherent distributed parallel memory. In *System Sciences, 1993, Proceeding of the Twenty-Sixth Hawaii International Conference on*, volume 2, pages 164–173. IEEE, 1993.

[53] Abdelsalam A Heddaya. *Coherence, non-coherence and Local Consistency in distributed shared memory for parallel computing.* Citeseer, 1992.

[54] Monika Rauch Henzinger, Valerie King, and Tandy Warnow. Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology. *Algorithmica*, 24(1):1–13, 1999.

[55] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[56] Phillip W Hutto and Mustaque Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, pages 302–309. IEEE, 1990.

[57] Liviu Iftode, Jaswinder Pal Singh, and Kai Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 277–287. ACM, 1996.

[58] Vadim Iosevich and Assaf Schuster. Distributed shared memory: To relax or not to relax? In Marco Danelutto, Marco Vanneschi, and Domenico Laforenza, editors, *Euro-Par 2004 Parallel Processing*, volume 3149 of *Lecture Notes in Computer Science*, pages 198–205. Springer Berlin Heidelberg, 2004.

[59] Andrey Kan, Jeffrey Chan, James Bailey, and Christopher Leckie. A query based approach for mining evolving graphs. In *Proceedings of the Eighth Australasian Data Mining Conference - Volume 101*, AusDM '09, pages 139–150, Darlinghurst, Australia, Australia, 2009. Australian Computer Society, Inc.

[60] U Kang, Duen Horng, and Christos Faloutsos. Inference of beliefs on billion-scale graphs. In *Large-scale Data Mining: Theory and Applications*, 2010.

[61] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998.

[62] George Karypis, Kirk Schloegel, and Vipin Kumar. Parmetis: Parallel graph partitioning and sparse matrix ordering library. *Dept. of Computer Science, University of Minnesota*, 1997.

[63] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 13–21, New York, NY, USA, 1992. ACM.

[64] Peter J Keleher, Alan L Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *USENIX Winter*, volume 1994, pages 23–36, 1994.

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

[66] Sai Charan Koduru, Min Feng, and Rajiv Gupta. Programming large dynamic data structures on a dsm cluster of multicores. In *7th International Conference on PGAS Programming Models*, volume 126, 2013.

[67] Leonidas Kontothanassis, Robert Stets, Galen Hunt, Umit Rencuzogullari, Gautam Altekar, Sandhya Dwarkadas, and Michael L. Scott. Shared memory computing on clusters with symmetric multiprocessors and system area networks. *ACM Trans. Comput. Syst.*, 23(3):301–335, August 2005.

[68] Anders Kristensen and Colin Low. Problem-oriented object memory: customizing consistency. In *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA '95, pages 399–413, New York, NY, USA, 1995. ACM.

[69] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 211–222, New York, NY, USA, 2007. ACM.

[70] Jérôme Kunegis. Konect: The koblenz network collection. In *Proceedings of the 22Nd International Conference on World Wide Web Companion*, WWW '13 Companion, pages 1343–1350, Republic and Canton of Geneva, Switzerland, 2013. International World Wide Web Conferences Steering Committee.

[71] Amlan Kusum, Keval Vora, Rajiv Gupta, and Iulian Neamtiu. Efficient Processing of Large Graphs via Input Reduction. In *ACM HPDC*, 2016.

[72] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *WWW*, pages 591–600, 2010.

[73] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: large-scale graph computation on just a pc. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, 2012.

[74] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[75] Leslie Lamport. How to make a mulitprocessor computer that correctly executes multiprocess programs. In *Readings in computer architecture*, pages 574–575. Morgan Kaufmann Publishers Inc., 2000.

[76] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 148–159, New York, NY, USA, 1990. ACM.

[77] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[78] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael 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.

[79] Michael Ley. The dblp computer science bibliography: Evolution, research issues, perspectives. In *SPIRE*, pages 1–10, 2002.

[80] Wen-Yew Liang, Chun ta King, and Feipei Lai. Adsmith: An efficient object-based distributed shared memory system on pvm. In *PVM. Proceedings of the 1996 International Symposium on Parallel Architecture (ISPAN 96*, pages 173–179. Press, 1996.

[81] Ee-Peng Lim, Viet-An Nguyen, Nitin Jindal, Bing Liu, and Hady Wirawan Lauw. Detecting product review spammers using rating behaviors. In *CIKM*, pages 939–948, 2010.

[82] Ilya Lipkind, Igor Pechtchanski, and Vijay Karamcheti. Object views: language support for intelligent object caching in parallel and distributed computations. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '99, pages 447–460, New York, NY, USA, 1999. ACM.

[83] Xin Liu and Tsuyoshi Murata. Advanced modularity-specialized label propagation algorithm for detecting communities in networks. *Physica A: Statistical Mechanics and its Applications*, 389(7):1493–1500, 2010.

[84] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012.

[85] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[86] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, Grzegorz Czajkowski, and Google Inc. Pregel: A system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[87] D. Manivannan and M. Singhal. Quasi-synchronous checkpointing: Models, characterization, and classification. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):703–713, July 1999.

[88] Paolo Massa and Paolo Avesani. Controversial users demand local trust metrics: An experimental study on epinions.com community. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 1*, AAAI'05, pages 121–126. AAAI Press, 2005.

[89] David Mosberger. Memory consistency models. *ACM SIGOPS Operating Systems Review*, 27(1):18–26, 1993.

[90] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *SOSP*, pages 439–455, 2013.

[91] Kamran Najeebullah, Kifayat Ullah Khan, Waqas Nawaz, and Young-Koo Lee. Bishard parallel processor: A disk-based processing engine for billion-scale graphs. *Journal of Multimedia & Ubiquitous Engineering*, 9(2):199–212, 2014.

[92] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, pages 456–471, 2013.

[93] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM.

[94] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford University, 1998.

[95] Russell Power and Jinyang Li. Piccolo: Building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 293–306, Berkeley, CA, USA, 2010. USENIX Association.

[96] Mayank Pundir, Luke M. Leslie, Indranil Gupta, and Roy H. Campbell. Zorro: Zero-cost reactive failure recovery in distributed graph processing. In *SoCC*, pages 195–208, 2015.

[97] Frederick Reiss, Kurt Stockinger, Kesheng Wu, Arie Shoshani, and Joseph M. Hellerstein. Enabling real-time querying of live and historical stream data. In *Proceedings of the 19th International Conference on Scientific and Statistical Database Management*, SSDBM '07, pages 28–, Washington, DC, USA, 2007. IEEE Computer Society.

[98] Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. On querying historical evolving graph sequences, 2011.

[99] Jason Riedy and Henning Meyerhenke. Scalable algorithms for analysis of massive, streaming graphs, 2012.

[100] Liam Roditty and Uri Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. *SIAM Journal on Computing*, 45(3):712–733, 2016.

[101] Ryan A. Rossi, Brian Gallagher, Jennifer Neville, and Keith Henderson. Modeling dynamic behavior in large evolving graphs. In *WSDM*, pages 667–676, 2013.

[102] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *SOSP*, pages 410–424, 2015.

[103] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: Edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488, 2013.

[104] Pratanu Roy, Arijit Khan, and Gustavo Alonso. Augmented sketch: Faster and more accurate stream processing. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1449–1463, New York, NY, USA, 2016. ACM.

[105] Semih Salihoglu and Jennifer Widom. GPS: A graph processing system. In *SSDBM*, pages 22:1–22:12, 2013.

[106] Lipton Richardand Jonathan S Sandberg. Pram: A scalable shared memory. Technical report, Technical Report TR-180-88, Princeton Univ., Dept. Comp. Sci, 1988.

[107] Daniel J. Scales and Kourosh Gharachorloo. Design and performance of the shasta distributed shared memory protocol. In *ICS*, pages 245–252, 1997.

[108] Martin Schulz, Jie Tao, and Wolfgang Karl. Improving the scalability of shared memory systems through relaxed consistency. In *Proceedings of the Second Workshop on Caching, Coherence, and Consistency (WC302)*, 2002.

[109] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 505–516, New York, NY, USA, 2013. ACM.

[110] Xiaowei Shen, Arvind, and Larry Rudolph. Cachet: an adaptive cache coherence protocol for distributed shared-memory systems. In *Proceedings of the 13th international conference on Supercomputing*, ICS '99, pages 135–144, New York, NY, USA, 1999. ACM.

[111] Yanyan Shen, Gang Chen, H. V. Jagadish, Wei Lu, Beng Chin Ooi, and Bogdan Marius Tudor. Fast failure recovery in distributed graph processing systems. *Proc. VLDB Endow.*, 8(4):437–448, December 2014.

[112] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. Tornado: A system for real-time iterative analysis over evolving data. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 417–430, New York, NY, USA, 2016. ACM.

[113] Yossi Shiloach and Shimon Even. An on-line edge-deletion problem. *Journal of the ACM (JACM)*, 28(1):1–4, 1981.

[114] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *PPoPP*, pages 135–146, 2013.

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

[116] Aman Singla, Umakishore Ramachandran, and Jessica Hodgins. Temporal notions of synchronization and consistency in beehive. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 211–220. ACM, 1997.

[117] Stack exchange inc. stack exchange data explorer: http://data.stackexchange.com/.

[118] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230. ACM, 2012.

[119] Jimeng Sun, Christos Faloutsos, Spiros Papadimitriou, and Philip S. Yu. GraphScope: Parameter-free mining of large time-evolving graphs. In *KDD*, pages 687–696, 2007.

[120] Toyotaro Suzumura, Shunsuke Nishii, and Masaru Ganse. Towards large-scale graph stream processing platform. In *WWW Companion*, pages 1321–1326, 2014.

[121] Lubos Takac and Michal Zabovsky. Data analysis in public social networks. In *International Scientific Conference and International Workshop Present Day Trends of Innovations*, pages 1–6, 2012.

[122] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A system for distributed graph mining. In *SOSP*, pages 425–440, 2015.

[123] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From" think like a vertex" to" think like a graph. *PVLDB*, 7(3):193–204, 2013.

[124] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[125] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 333–342. ACM, 2014.

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

[127] Keval Vora, Rajiv Gupta, and Guoqing Xu. Synergistic analysis of evolving graphs. *ACM Transactions on Architecture and Code Optimization*, 13(4):32:1–32:27, October 2016.

[128] Keval Vora, Rajiv Gupta, and Guoqing Xu. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 237–251, New York, NY, USA, 2017. ACM.

[129] Keval Vora, Sai Charan Koduru, and Rajiv Gupta. ASPIRE: Exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based DSM. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 861–878, New York, NY, USA, 2014. ACM.

[130] Keval Vora, Chen Tian, Rajiv Gupta, and Ziang Hu. Coral: Confined recovery in distributed asynchronous graph processing. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 223–236, New York, NY, USA, 2017. ACM.

[131] Keval Vora, Guoqing Xu, and Rajiv Gupta. Load the edges you need: A generic i/o optimization for disk-based graph processing. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 507–522, Berkeley, CA, USA, 2016. USENIX Association.

[132] Guozhang Wang, Wenlei Xie, Alan Demers, and Johannes Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, 2013.

[133] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. Graspan: A single-machine disk-based graph system for interprocedural static analyses of

large-scale systems code. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 389–404, New York, NY, USA, 2017. ACM.

[134] Kai Wang, Guoqing Xu, Zhendong Su, and Yu David Liu. GraphQ: Graph query processing with abstraction refinement—programmable and budget-aware analytical queries over very large graphs on a single PC. In *USENIX ATC*, pages 387–401, 2015.

[135] Peng Wang, Kaiyuan Zhang, Rong Chen, and Haibo Chen. Replication-based fault-tolerance for large-scale graph processing. In *IEEE/IFIP DSN*, pages 562–573, 2014.

[136] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of /'small-world/' networks. *Nature*, 393(6684):440–442, 06 1998.

[137] Yahoo! Webscope Program. `http://webscope.sandbox.yahoo.com/`.

[138] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[139] John W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17(9):530–531, September 1974.

[140] Byung-Hyun Yu, Zhiyi Huang, Stephen Cranefield, and Martin Purvis. Homeless and home-based lazy release consistency protocols on distributed shared memory. In *Proceedings of the 27th Australasian Conference on Computer Science - Volume 26*, ACSC '04, pages 117–123, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[141] Mindi Yuan, Kun-Lung Wu, Gabriela Jacques-Silva, and Yi Lu. Efficient processing of streaming graphs for evolution-aware clustering. In *CIKM*, pages 319–328, 2013.

[142] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX NSDI*, pages 2–2, 2012.

[143] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. *HotCloud*, 10:10–10, 2010.

[144] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Presented as part of the*, 2012.

[145] Erik Zeitler and Tore Risch. Massive scale-out of expensive continuous queries, 2011.

[146] ZeroMQ. http://zeromq.org/.

[147] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Pruning dynamic slices with confidence. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 169–180, 2006.

[148] Xiangyu Zhang and Rajiv Gupta. Matching execution histories of program versions. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 197–206, New York, NY, USA, 2005. ACM.

[149] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Precise dynamic slicing algorithms. In *Proceedings of the 25th International Conference on Software Engineering*, pages 319–329, 2003.

[150] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. FlashGraph: processing billion-node graphs on an array of commodity ssds. In *FAST*, pages 45–58, 2015.

[151] Yuanyuan Zhou, Liviu Iftode, Jaswinder Pal Sing, Kai Li, Brian R. Toonen, Ioannis Schoinas, Mark D. Hill, and David A. Wood. Relaxed consistency and coherence granularity in dsm systems: a performance evaluation. In *Proceedings of the sixth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '97, pages 193–205, New York, NY, USA, 1997. ACM.

[152] Xiaojin Zhu and Zoubin Ghahramani. Learning from labeled and unlabeled data with label propagation. Technical Report CALD-02-107, Carnegie Mellon University, 2002.

[153] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX ATC*, pages 375–386, 2015.