On Improving Distributed Pregel-like Graph Processing Systems

by

Minyang Han

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2015

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The considerable interest in distributed systems that can execute algorithms to process large graphs has led to the creation of many graph processing systems. However, existing systems suffer from two major issues: (1) poor performance due to frequent global synchronization barriers and limited scalability; and (2) lack of support for graph algorithms that require serializability, the guarantee that parallel executions of an algorithm produce the same results as some serial execution of that algorithm.

Many graph processing systems use the bulk synchronous parallel (BSP) model, which allows graph algorithms to be easily implemented and reasoned about. However, BSP suffers from poor performance due to stale messages and frequent global synchronization barriers. While asynchronous models have been proposed to alleviate these overheads, existing systems that implement such models have limited scalability or retain frequent global barriers and do not always support graph mutations or algorithms with multiple computation phases. We propose barrierless asynchronous parallel (BAP), a new computation model that overcomes the limitations of existing asynchronous models by reducing both message staleness and global synchronization while retaining support for graph mutations and algorithms with multiple computation phases. We present GiraphUC, which implements our BAP model in the open source distributed graph processing system Giraph, and evaluate it at scale to demonstrate that BAP provides efficient and transparent asynchronous execution of algorithms that are programmed synchronously.

Secondly, very few systems provide serializability, despite the fact that many graph algorithms require it for accuracy, correctness, or termination. To address this deficiency, we provide a complete solution that can be implemented on top of existing graph processing systems to provide serializability. Our solution formalizes the notion of serializability and the conditions under which it can be provided for graph processing systems. We propose a partition-based synchronization technique that enforces these conditions efficiently to provide serializability. We implement this technique into Giraph and GiraphUC to demonstrate that it is configurable, transparent to algorithm developers, and more performant than existing techniques.

Acknowledgements

I owe much thanks to my supervisor Khuzaima Daudjee, whose careful guidance and scrupulous attention to detail both sharpened my communication skills and made this thesis possible. I am also grateful to my readers Tamer Özsu and Ihab Ilyas for their insightful questions and comments. Finally, I'd like to thank colleagues, both IQC and DB, as well as my family and friends for their support during my time at Waterloo.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Due to the wide variety of real-world problems that rely on processing large amounts of graph data, graph data processing has become ubiquitous. For example, web graphs containing over 60 trillion indexed webpages must be processed by Google's ranking algorithms to determine influential vertices [35]. Massive social graphs are processed at Facebook to compute popularity and personalized rankings, determine shared connections, find communities, and propagate advertisements for over 1 billion monthly active users [30]. Scientists are also leveraging biology graphs to understand protein interactions [58] and cell graphs for automated cancer diagnosis [37].

Graph processing solves real-world problems through graph algorithms that are implemented and executed on *graph processing systems*. These systems provide programming and computation models, which developers use to implement such algorithms, as well as any correctness guarantees that algorithms require. The systems are also used to execute algorithms against desired input graphs.

Google's Pregel [54] is one such system that provides a native graph processing API by pairing the bulk synchronous parallel (BSP) computation model [73] with a vertex-centric, or "think like a vertex", programming model. This has inspired many popular open source *Pregel-like* graph processing systems, including Apache Giraph [1] and GraphLab [50].

However, these Pregel-like systems suffer from two major issues: (1) poor performance at scale due to the frequent use of global synchronization barriers and (2) a lack of serializability guarantees for graph algorithms.

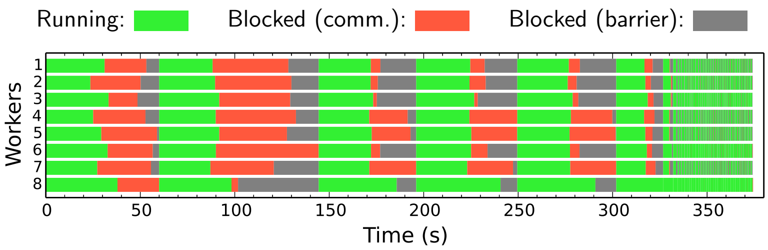


Figure 1.1: Communication and synchronization overheads for a BSP execution of weakly connected components using 8 worker machines on TW (Table 3.1).

1.1 Performance

For graph processing systems, one key systems-level performance concern stems from the strong isolation, or staleness, of messages in the synchronous BSP model. Relaxing this isolation enables asynchronous execution, which allows vertices to see up-to-date data and leads to faster convergence and shorter computation times [50]. For *pull-based* systems such as GraphLab, where vertices pull data from their neighbours on demand, asynchrony is achieved by combining GraphLab's Gather, Apply, Scatter (GAS) model with distributed locking (which prevents data races). For *push-based* systems such as Giraph [1], Giraph++ [72], and GRACE [74], where vertices explicitly push data to their neighbours as messages, asynchrony is achieved through the *asynchronous parallel* (AP) model. The AP model extends BSP and avoids distributed locking, which is advantageous as distributed locking is difficult to tune and was observed to incur substantial communication overheads, leading to poor horizontal scalability (i.e., scale out) [39].

A second key performance concern is the frequent use of global synchronization barriers in the BSP model. These global barriers incur costly communication and synchronization overheads and also give rise to the straggler problem, where fast machines are blocked waiting for a handful of slow machines to arrive at the barrier. For example, Figure 1.1 illustrates an actual BSP execution of the weakly connected components algorithm in which workers are, on average, blocked on communication and synchronization for 46% of the total computation time. GraphLab's asynchronous mode (GraphLab async) attempts to avoid this blocking by removing global barriers altogether and instead relying on distributed locking. However, as pointed out above, this solution scales poorly and can result in even greater communication overheads. Although the AP model avoids distributed locking, it relies on the use of frequent global barriers and thereby suffers from the same overheads as the BSP model.

Finally, there is a third concern of usability and also compatibility. The simple and deterministic nature of the BSP model enables algorithm developers to easily reason about, and debug, their code. In contrast, a fully exposed asynchronous model requires careful consideration of the underlying consistency guarantees as well as coding and debugging in a non-deterministic setting, both of which can be confusing and lead to buggy code. Hence, a performant graph processing system should allow developers to code for the BSP model and transparently execute with an efficient asynchronous computation model. Existing systems that provide asynchronous execution leak too much of the underlying asynchronous mechanisms to the developer API [74], impeding usability, and do not support algorithms that require graph mutations [50, 74] or algorithms with multiple computation phases [72], impeding compatibility.

To address these concerns, we propose a *barrierless asynchronous parallel* (BAP) computation model that both relaxes message isolation and substantially reduces the frequency of global barriers, without using distributed locking. Our system, GiraphUC, implements the proposed BAP model in Giraph, a popular open source system. GiraphUC preserves Giraph's BSP developer interface and fully supports algorithms that perform graph mutations or have multiple computation phases. GiraphUC is also more scalable than GraphLab async and achieves good performance improvements over Giraph, Giraph async (which uses the AP model), and GraphLab sync and async. Thus, GiraphUC enables developers to code for a synchronous BSP model and transparently execute with an asynchronous BAP model to maximize performance.

1.2 Serializability

A key guarantee that graph processing systems can provide is *serializability*. Informally, a graph processing system provides serializability if it can guarantee that parallel executions of an algorithm, implemented with its programming and computation models, produce the same results as some serial execution of that algorithm [34].

Serializability is required by many algorithms, for example in machine learning, to provide both theoretical and empirical guarantees for convergence or termination. Parallel algorithms for combinatorial optimization problems experience a drop in performance and accuracy when parallelism is increased without consideration for serializability. For example, the Shotgun algorithm for L_1-regularized loss minimization parallelizes sequential coordinate descent to handle problems with high dimensionality or large sample sizes [19]. As the number of parallel updates is increased, convergence is achieved in fewer iterations. However, after a sufficient amount of parallelism, divergence occurs and *more* iterations are required to reach convergence [19]. Similarly, for energy minimization on NK energy functions (which model a system of discrete spins), local search techniques experience an abrupt degradation in the solution quality as the number of parallel updates is increased [67]. Other algorithms such as dynamic alternating least squares (ALS) have unstable accuracy when executed without serializability [50], while Gibbs sampling requires serializability for statical correctness [33]. Graph coloring requires serializability to terminate on dense graphs [34] and, even for sparse graphs, will use significantly fewer colors and complete in only a single iteration when executed serializably.

Providing serializability in a graph processing system is fundamentally a system-level problem that informally requires: (1) vertices see up-to-date data from their neighbours and (2) no two neighbouring vertices execute concurrently. The general approach is to pair an existing system or computation model with a synchronization technique that enforces conditions (1) and (2). Despite this, of the numerous graph processing systems that have appeared over the past few years, few provide serializability as a configurable option. For example, popular systems like Pregel [54], Giraph [1], and GPS [63] do not provide serializability.

Giraphx [71] provides serializability by pairing one of two synchronization techniques, *token passing* and *vertex-based distributed locking*, with the AP model. However, it implements these two techniques as part of specific algorithms rather than within the system. Consequently, Giraphx unnecessarily couples and exposes internal system details to user algorithms, meaning serializability is neither a configurable option nor transparent to the algorithm developer. Furthermore, its implementation of vertex-based distributed locking unnecessarily divides each superstep, an iteration of computation, into multiple sub-supersteps in which only a subset of vertices can execute. This exacerbates the already expensive communication and synchronization overheads associated with the global synchronization barriers that occur at the end of each superstep (Chapter 1.1), resulting in poor performance.

GraphLab [50], which now subsumes PowerGraph [34], takes a different approach by starting with its asynchronous mode (GraphLab async), which avoids expensive global barriers by using distributed locking. GraphLab async provides the option to execute with or without serializability and uses vertex-based distributed locking as its synchronization technique. However, GraphLab async suffers from high communication overheads (Chapter 1.1) and scales poorly with this synchronization technique.

Irrespective of the specific system, synchronization techniques used to enforce conditions (1) and (2) fall on a spectrum that trades off parallelism with communication overheads (Figure 1.2). In particular, token passing and vertex-based distributed locking fall on the

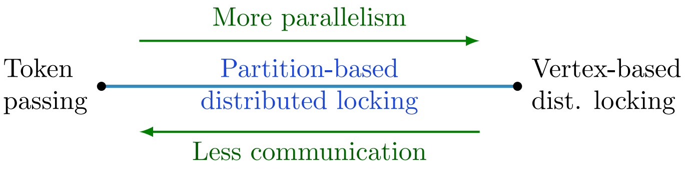

More parallelism
Token passing
Partition-based distributed locking
Vertex-based dist. locking
Less communication

Figure 1.2: Spectrum of synchronization techniques.

extremes of this spectrum: token passing uses minimal communication but unnecessarily restricts parallelism, forcing only one machine to execute at a time, while vertex-based distributed locking uses a dining philosopher algorithm to maximize parallelism but incurs substantial communication overheads due to every vertex needing to acquire forks from their neighbours.

To overcome these issues, we first formalize the notion of serializability in graph processing systems and prove that the above techniques ensure serializability. To the best of our knowledge, no existing work has presented such a formalization. To address the shortcomings of the existing techniques, we propose a novel *partition-based distributed locking* solution that allows control over the coarseness of locking and the resulting trade-off between parallelism and communication overheads (Figure 1.2). We implement all three techniques at the system level, in Giraph async and GiraphUC (Chapter 1.1), so that they are performant, configurable, and transparent to algorithm developers. We demonstrate through experimental evaluation that our partition-based solution substantially outperforms the existing techniques.

1.3 Thesis Outline

We introduce the BSP, AP, and GAS computation models, categorize the Pregel-like graph processing systems that use them, and discuss related work in Chapter 2. In Chapter 3, we describe the BAP model and its implementation in Giraph, GiraphUC [38]. In Chapter 4, we formalize serializability in Pregel-like graph processing systems and introduce our partition-based technique before concluding with future work in Chapter 5.

Chapter 2

Related Work

We begin by introducing the BSP, AP, and GAS computation models in Chapter 2.1, followed by a categorization of Pregel-like graph processing systems in Chapter 2.2, and conclude with a discussion on some additional systems related to graph processing.

2.1 Existing Computation Models

2.1.1 BSP Model

Bulk synchronous parallel (BSP) [73] is a parallel computation model in which computations are divided into a series of (BSP) *supersteps* separated by global barriers (Figure 2.1). To support iterative graph computations, Pregel (and Giraph) pairs BSP with a vertex-centric programming model, in which vertices are the units of computation and edges act as communication channels between vertices.

Graph computations are specified by a user-defined compute function that executes, in parallel, on all vertices in each superstep. Consider, as a running example, the BSP execution of the weakly connected components (WCC) algorithm (Figure 2.2). In the first superstep, each vertex initializes its vertex value to a component ID. In the subsequent supersteps, it updates this value with any smaller IDs received from its in-edge neighbours and propagates any changes along its out-edges (Chapter 3.4.2.2). Crucially, messages sent in one superstep are visible only during the next superstep. For example, ID 1 sent by v_2 in superstep 2 is seen by v_3 only in superstep 3. At the end of each superstep, all vertices "vote to halt" to become inactive. A vertex is reactivated by incoming messages,

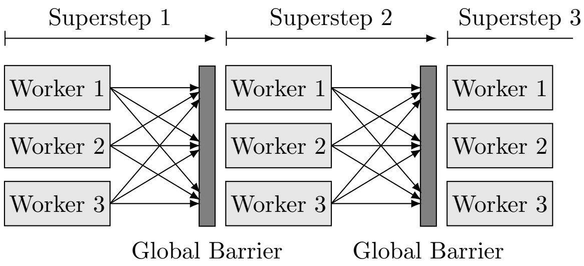


Figure 2.1: The BSP model, illustrated with three supersteps and three workers [45].

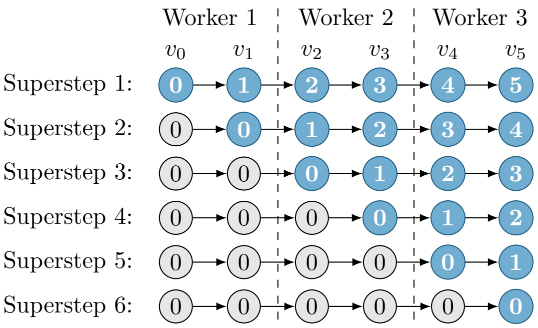


Figure 2.2: BSP execution of a WCC example. Gray vertices are inactive. Blue vertices have updated vertex values.

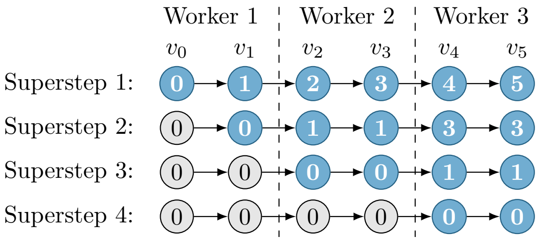


Figure 2.3: AP execution of the WCC example.

for example v_1 in superstep 2. The computation terminates at the end of superstep 6 when all vertices are inactive and no more messages are in transit.

Pregel and Giraph use a master/workers model. The master machine partitions the graph among the worker machines, coordinates all global barriers, and performs termination checks based on the two aforementioned conditions. BSP uses a push-based approach, as messages are pushed by the sender and buffered at the receiver. Finally, as described in Chapter 1.2, the BSP model does not provide serializability. We demonstrate this with a concrete example in Chapter 4.1.1.

2.1.2 AP Model

The asynchronous parallel (AP) model improves on the BSP model by reducing the staleness of messages. It allows vertices to immediately see their received messages instead of delaying them until the next superstep. These messages can be *local* (sent between vertices owned by a single worker) or *remote* (sent between vertices of different workers). The AP model retains global barriers to separate supersteps, meaning that messages that do not arrive in time to be seen by a vertex in superstep i (i.e., because the vertex has already been executed) will become visible in the next superstep $i+1$. GRACE and, to a lesser extent, Giraph++'s hybrid mode both implement the AP model (Chapter 3.3.2).

To see how reducing staleness can improve performance, consider again our WCC example from Figure 2.2. For simplicity, assume that workers are single-threaded so that they execute their two vertices sequentially. Then, in the BSP execution (Figure 2.2), v_3 in superstep 3 sees only a stale message with the ID 1, sent by v_2 in superstep 2. In contrast, in the AP execution (Figure 2.3), v_3 sees a newer message with the ID 0, sent from v_2 in superstep 3, which enables v_3 to update to (and propagate) the component ID 0 earlier. Consequently, the AP execution is more efficient as it completes in fewer supersteps than the BSP execution.

However, the AP model suffers from communication and synchronization overheads, due to retaining frequent global barriers, and has limited algorithmic support (Chapter 3.1). Furthermore, like BSP, the AP model does not provide serializability (Chapter 4.1.2).

2.1.3 GAS Model

The Gather, Apply, and Scatter (GAS) model is used by GraphLab for both its synchronous and asynchronous modes, which we refer to as GraphLab sync and GraphLab async. These two system modes implement the sync GAS and async GAS models, respectively.

As its name suggests, the GAS model consists of three phases. In the gather phase, a vertex accumulates (pulls) information about its neighbourhood; in the apply phase, the vertex applies the accumulated data to its own value; and in the scatter phase, the vertex updates its adjacent vertices and edges and activates neighbouring vertices. Like Pregel and Giraph, GraphLab pairs GAS with a vertex-centric programming model. However, as evidenced by the Gather phase, GAS is pull-based rather than push-based.

Sync GAS is similar to BSP: vertices are executed in supersteps separated by global barriers and the effects of apply and scatter of one superstep are visible only to the gather of the next superstep. Hence, like BSP, sync GAS does not provide serializability as vertices can only use information from the previous superstep. For example, WCC would proceed similarly to Figure 2.2. Lastly, unlike BSP, vertices are inactive by default and must be explicitly activated by a neighbour during scatter to be executed in the next superstep.

Async GAS is different from AP as it has no notion of supersteps. To execute a vertex u, each GAS phase individually acquires a write lock on u and read locks on u's neighbours to prevent data races [8]. GraphLab async implements async GAS by maintaining a large pool of lightweight threads (called fibers) and pairing each fiber with an available vertex. Before executing each GAS phase, the fiber acquires the necessary locks through distributed locking. To terminate, GraphLab async runs a distributed consensus algorithm to check that all workers' schedulers are empty (i.e., no more vertices to execute).

In contrast, AP can avoid async GAS's expensive distributed locking because it is push-based: messages are received only after a vertex finishes its computation and explicitly pushes such messages. Since messages are buffered in the recipient machine's local message store, concurrent reads and writes to the store (i.e., data races) can be handled with local locks or lock-free data structures. Furthermore, the AP model can rely on the master to check for termination, which avoids the overheads of a distributed consensus algorithm.

Finally, async GAS does not provide serializability because the GAS phases of different vertex computations can interleave [34]. To provide serializability (Chapter 4.2.5), a synchronization technique must be added on top of async GAS. This technique prevents neighbouring computations from interleaving by performing vertex-based distributed locking over all three GAS phases. Note that this is different from the per-phase distributed locking used by async GAS to prevent data races. As we show in Chapter 4.2, async GAS provides serializability when paired with this synchronization technique.

2.2 Categorization

Graph processing systems can be categorized based on their developer API and type of system execution (Table 2.2) as well as general feature support (Table 2.3).

The developer API consists of the *programming* and *computation models* that the system exposes to algorithm developers for creating algorithms. For example, the API exposed by Pregel [54] is a vertex-centric programming model paired with a BSP computation model.

The type of system execution depends on both the computation model used by the system, which may be different from what is exposed by its API, and additional properties of the system, some of which are intrinsic to the computation model used. For example, if the system uses the BSP model then it must have global barriers, as barriers are intrinsic to BSP. On the other hand, providing serializability is not intrinsic to any computation model. Most systems use a computation model that is identical to the one exposed for its developer API. However, this need not always be the case: GiraphUC, for example, exposes a BSP (or AP) computation model for developers but transparently executes using the BAP computation model (Chapter 3.2).

Lastly, a system's general feature support consists of its *graph partitioning* and *dynamic migration* (re-partitioning) schemes, message optimizations, and support for algorithms that perform *graph mutations* (addition and deletion of vertices and edges) and algorithms with multiple computation phases (*multi-phase algorithms*).

We break down and detail programming models, computational models, and each of the general feature supports next.

2.2.1 Programming Models

Most systems are *vertex-centric* (v-centric), following the "think like a vertex" programming model used by Pregel and described in Chapter 2.1.1. For example, Giraph [1], GPS [63], GraphLab [50], and GiraphUC are all vertex-centric (Table 2.2).

There are also systems that consider a *partition-centric* (p-centric) approach, where developers define the execution of each partition of vertices (where each worker machine can have multiple partitions). For example, in addition to the regular vertex-centric compute function, Blogel [80] lets developers specify a B-compute function that is executed on each block (partition) of vertices. In Giraph++ [72], developers define only a compute function for executing partitions of vertices. This improves expressibility by allowing developers to execute sequential algorithms over the vertices of each partition. However, it negatively impacts usability as the algorithms are more complex: since partitions execute in parallel, developers must reason about the coordination of each partition's boundary vertices because they must communicate with neighbours that belong to other partitions.

Finally, the single-machine disk-based system X-Stream [62] uses an *edge-centric* approach: developers define scatter and gather-and-apply functions that are performed on edges rather than vertices. X-Stream uses this to avoid random I/O, which leads to better performance because portions of the graph must be streamed from disk back into memory during computation. However, the edge-centric approach is less intuitive and is thus not widely adopted: other single-machine disk-based systems like FlashGraph [86], GraphChi [48], TurboGraph [40], and VENUS [24] all use a vertex-centric programming model. Since our focus is on in-memory systems, we exclude these disk-based systems from Tables 2.2 and 2.3. They are discussed in Chapter 2.4.

2.2.2 Computation Models

Computation models can be broken down and categorized based on their core features. Some of these features are intrinsic to the model (i.e., the model is defined to have it), while others are extrinsic and depend on whether the graph processing system implements it. For example, the BSP computation model will, by definition, always have global barriers. In contrast, the GAS model can be implemented either as a synchronous model or as an asynchronous model and thus with or without global barriers.

Computation models have three central pillars: *synchrony*, *push/pull*, and *global barriers*. We additionally have an orthogonal but equally important pillar of *serializability*. Serializability is closely related to synchrony but can be thought of as orthogonal to the computation model: one can provide serializability in a model-agnostic manner[1] on top of existing computation models (Chapter 4.2.5).

While the design choices are targeted to improve system performance independently for each pillar, these choices may also indirectly impact the performance of other pillars. For example, although asynchronous models generally perform better than synchronous models, async GAS's support for a pull-based approach leads to significant communication overheads due to distributed locking (Chapter 2.1.3), ultimately resulting in poor performance. In the case of providing serializability, asynchronous computation models can support a wider range of synchronization techniques, including a more performant partition-based distributed locking technique (Chapter 1.2), which substantially increases their performance advantage over synchronous computation models.

2.2.2.1 Synchrony

As described in Chapters 1.1 and 2.1, a computation model is *synchronous* (sync) if all updates of any vertex u in superstep i is visible to other vertices only in superstep $i + 1$. That is, recipients do not see the messages sent (pushed) by u until superstep $i + 1$. Equivalently, for pull-based models, vertices do not see an up-to-date value when pulling u's vertex value.

A model is *asynchronous* (async) when this is relaxed: i.e., the update of a vertex u can be seen by other vertices in the same superstep. Importantly, asynchronous models do not guarantee that other vertices will always see an up-to-date vertex u—it only removes the restriction that says they cannot. For example, in the AP model (Chapter 2.1.2),

[1] However, synchronous computation models support only a subset of synchronization techniques, which provide serializability, whereas asynchronous models have no such limitations (Chapter 4.3.1.1).

vertices of worker machine W_1 will immediately see changes to other vertices belonging to W_1 (assuming sequential execution) but can fail to see changes to vertices of W_2 due to network delays or parallel execution.

Systems can also implement computation models in a way that falls between sync and async. Blogel and Giraph++ allow vertex updates to be immediately visible to vertices of the same partition but delay messages between different partitions until the next superstep. Similarly for Giraph++'s hybrid mode, which implements the AP model for the vertex-centric setting. Consequently, these systems implement an AP model that is *partial async*. On the other hand, PowerSwitch [75] is a system, built on top of GraphLab, that dynamically switches between the sync and async GAS models by predicting the throughput (number of vertices processed per unit time) of each mode. Thus, it is both sync and async.

Our definition follows the standard use of "synchronous" and "asynchronous" in existing literature. Notably, we distinguish between synchrony and the presence of global barriers. This is in contrast to a recent survey [55] that confusingly uses "asynchronous" to refer to async models without global barriers and "hybrid" for async models with global barriers.

2.2.2.2 Push vs. Pull

Per Chapter 2.1, a model is push-based if vertices explicitly send (push) messages to other vertices. A model is pull-based if vertices pull data from other vertices. In general, systems tend to support either push only (e.g., Pregel, Giraph, GiraphUC) or both push and pull (e.g., GraphLab). When a system supports both push and pull, its design becomes constrained by the need to support pull. For example, vertices must use distributed locking to prevent data races and ensure they pull consistent data (i.e., data that is not being simultaneously updated). Furthermore, asynchronous pull-based systems cannot easily batch messages, as pulls can occur at any time, so there are fewer opportunities for message optimizations compared to a push-based system. This limitation is why we refer to GraphLab as a pull-based system, despite it also supporting push.

2.2.2.3 Global Barriers

Whether or not global barriers are used is generally intrinsic to the particular computation model. For example, BSP, AP, and sync GAS all use frequent global barriers between every superstep. In contrast, async GAS uses no global barriers, while BAP uses the minimal number of global barriers necessary to enforce correctness (for algorithms written for the

BSP or AP models). As global barriers negatively impact performance (Chapter 1.1), its presence is indicated with a red ✓ in Table 2.2.

All systems provide a minimal level of consistency to ensure vertices do not, for example, read values that are simultaneously being modified. This is required because algorithm developers have no access to the internal system details and thus cannot manually prevent such data races. Consequently, systems that attempt to achieve better performance by removing all global barriers must replace them with some alternative consistency mechanism. As described in Chapter 2.1.3, GraphLab async (which implements async GAS) uses fine-grained distributed locking to replace global barriers. However, as outlined in Chapter 1.1, this approach incurs significant communication overheads and degrades performance. Hence, for Table 2.2, we indicate the absence of global barriers with an orange ✗.

2.2.2.4 Serializability

Lastly, while asynchronous models remove the synchronous models' restriction on seeing up-to-date vertex values, they do not provide any guarantees of always seeing such up-to-date values (Chapter 4.2). Serializability goes one step further to guarantee that every vertex sees up-to-date data from their neighbours, such that the parallel executions of an algorithm produce the same results as some serial execution of the same algorithm.

As discussed in Chapter 1.2, very few graph processing systems provide serializability (Table 2.2). Only GraphLab async and Giraphx [71] consider and implement serializability. GraphLab async provides serializability as a transparent and optional feature, while Giraphx implements it directly in user algorithms, which is neither transparent nor configurable. In Chapter 4, we show how to provide serializability for existing computation models and, in particular, implement transparent and configurable serializability support for Giraph async and GiraphUC.

2.2.3 Additional System Features

2.2.3.1 Graph Partitioning

Distributed graph processing systems must partition the input graph across multiple worker machines prior to performing computation. To do so, systems partition using either *edge-cuts* or *vertex-cuts*. For edge-cut, each vertex belongs to a single worker while an edge can span two workers. For vertex-cut, each edge belongs to a single worker while vertices can span multiple workers: for each split vertex u, one worker owns the *primary copy* of

u while all other workers owning a neighbour of u get a local read-only replica of u. The primary advantage of vertex-cut partitioning is that it splits high-degree vertices across multiple machines. Since many natural graphs follow a power-law degree distribution, this splitting provides better workload balance than edge-cut partitioning [34]. However, vertex-cut partitioning may add additional communication overheads as the system must synchronize all replicas with the primary copy for every split vertex [77].

All systems perform a static partitioning of the input graph prior to computation. All systems have random hash partitioning as their standard static partitioning algorithm (Table 2.3): each vertex or edge is hashed to a worker machine, for edge-cut or vertex-cut respectively. Systems such as Blogel and Giraph++, which offer partition-centric programming models, use partitioning algorithms that, compared to hash partitioning, produce lower edge-cuts but at the cost of much longer partitioning times. GraphLab also supports several additional static partitioning algorithms (Table 2.3). However, GraphLab's oblivious algorithm is much slower than hash partitioning, while its grid and PDS algorithms support only a number of machines that is perfect square and $p^2 + p + 1$ (for p prime) respectively. Similarly, GraphX [78] supports 2D hash partitioning but only for a perfect square number of machines. Finally, GRACE [74] is a single-machine system that did not initially perform any static partitioning on the input graph. However, it was found in [76] that performance can be dramatically improved by partitioning the graph into blocks (partitions) using METIS and associating computation threads with available partitions. A similar approach is used in distributed systems such as Giraph, where each worker owns multiple partitions and its computation threads are paired with available partitions (Chapter 3.3.1).

Some systems also feature dynamic migration, or re-partitioning, to improve workload balance and network usage during the computation. As dynamic migration is not the focus of this thesis, we briefly survey only some existing techniques. Additional details can be found in [55] or in the respective system's papers. GPS [63] performs re-partitioning by exchanging vertices between workers based on the amount of data sent by each vertex. Its scheme locates migrated vertices by relabelling their vertex IDs and updating the adjacency lists in which they appear. Consequently, the scheme does not work for algorithms such as DMST (Chapter 3.4.2.3) that must send messages to specific vertex IDs. The scheme decreases network I/O but does not always improve computation time [63, 39]. Similarly, CatchW [65] uses workload and message activity as heuristics to select vertices to migrate. Mizan [45] performs migration based on a z-score, calculated using the number of incoming and outgoing messages and the response time of machines. Its scheme pairs over and under-utilized workers to exchange vertices with outlier z-scores and locates migrated vertices using a distributed hash table lookup service. Finally, X-Pregel [12] migrates vertices based on the number of messages sent and received but restricts migrations to be performed by only one worker at a time to reduce communication overheads. However, like in GPS, the scheme decreases network I/O but increases computation time [12].

2.2.3.2 Message Optimizations

Graph processing systems use several optimizations to reduce the number of messages and thus communication overheads. There are three main types of message optimizations: *message combiner*, *receiver-side scattering*, and *message batching*.

Message combiners, as the name suggests, combine multiple messages to reduce communication and/or memory costs. A sender-side combiner combines all messages at a worker W_i destined for a vertex u into a single message, so that only one message needs to be sent from W_i to u's worker machine. A receiver-side combiner, in contrast, combines all received messages for u into a single message to save memory. In either case, the combiner is implemented by the algorithm developer and must be an associative and commutative operation as messages can arrive at any time [54]. Systems like Pregel [54] and Signal/-Collect [69] support both sender-side and receiver-side combining (Table 2.3). However, nearly all other systems support only receiver-side combining because sender-side combining tends to yield little benefit: there are insufficient opportunities for combining messages to offset the overheads of maintaining additional data structures to hold and combine outgoing messages [53, 63]. Additionally, asynchronous pull-based systems such as GraphLab async generally do not support sender-side combiners because messages between workers are difficult to batch (as they are pulled on demand rather than pushed).

Receiver-side scatter reduces communication costs by combining multiple messages sent by a vertex u of W_1 to its neighbours on W_2 into a single message from W_1 to W_2 [55]. This works only when u is broadcasting the same message to all of its neighbours. GPS implements this as an optional feature called Large Adjacency List Partitioning (LALP), which performs receiver-side scatter for all vertices whose edge degree is above a user-defined threshold. The threshold ensures that only high-degree vertices perform this optimization, as receiver-side scatter is of limited benefit (and may incur overheads) for low-degree vertices. Similarly, LFGraph [41] and X-Pregel [12] also provide receive-side scatter as an optimization.

Finally, message batching amortizes communication overheads by flushing large batches of messages to the network rather than sending each message individually. This substantially improves network performance and shortens computation times. All synchronous systems perform message batching as it is a trivial optimization: messages need not be

Table 2.1: Algorithmic requirements for graph processing and machine learning algorithms.

Algorithm	**Graph Mutations**	**Multi-phase**
Adamic-Adar similarity [9, 5]	✗	✓
Alternate least squares (ALS) [88, 50, 5]	✗	✗
Approximate maximum weight matching (MWM) [57, 64]	✓	✓
BFS/DFS [36]	✗	✗
Bipartite maximal matching (BMM) [54, 51]	✗	✓
Clustering coefficient [5]	✗	✓
Community detection/label propagation [60, 49, 74]	✗	✗
Diameter estimation [43]	✗	✗
Graph coarsening [72]	✓	✓
Graph coloring [34]	✗	✗
Jaccard similarity [5]	✗	✓
K-core [59, 5]	✓	✗
K-means [5]	✗	✓
Maximal independent sets [52, 64]	✓	✓
Maximum B-Matching [28, 5]	✗	✗
Minimum spanning tree (DMST) [26, 64, 39]	✓	✓
Multi-source shortest paths [5]	✗	✓
PageRank [54, 39]	✗	✗
Reachability [80]	✗	✗
Semi-clustering [54, 5]	✗	✗
Shiloach-Vishkin's algorithm (SV) [81, 51]	✗	✓
Single-source shortest path (SSSP) [27, 39]	✗	✗
Singular value decomposition (SVD++) [46, 5]	✓	✓
Stochastic gradient descent (SGD) [47, 5]	✓	✓
Strongly connected components (SCC) [56, 64]	✗	✓
Top-K PageRank [45]	✗	✗
Triangle finding [59, 5]	✗	✓
Weakly connected components (WCC) [44, 39]	✗	✗

delivered until after the global barrier anyway. Batching is much more difficult under asynchronous pull-based systems, due to the unpredictability of when a vertex will pull from its neighbours. Consequently, systems such as GraphLab async do not support message batching. In contrast, asynchronous push-based systems such as GiraphUC can perform message batching with ease as the sender can control when messages should be flushed to the network.

2.2.3.3 Algorithmic Support

Many popular graph processing and machine learning algorithms that developers want to execute require support for graph mutations and multiple computation phases (Table 2.1). Thus, it is imperative that graph processing systems fully support such algorithms.

Graph mutations is supported by a majority of systems, including Pregel, Giraph, and GiraphUC (Table 2.3). Other systems have only partial support for graph mutations: GraphLab and PowerSwitch cannot delete vertices or edges, while GPS and X-Pregel do not support vertex mutations. Finally, CatchW, GRACE [74], GraphX, LFGraph, and MOCgraph [87] do not support graph mutations at all.

Multi-phase algorithms are algorithms with multiple computation phases, each of which can have different computation logic. Nearly all systems in Table 2.3 support multi-phase algorithms. Giraph++'s hybrid mode, Giraphx, and MOCgraph's asynchronous option do not support multi-phase algorithms because messages from different computation phases can become mixed together, resulting in computation errors (Chapter 3.2.3). In contrast, GRACE allows developers to specify which version of messages to read (i.e., from the current superstep or previous superstep) and so it can support multi-phase algorithms with assistance from the developer. As Chapter 3.2.3 will describe, Giraph async and GiraphUC support multi-phase algorithms using only a very minor change to the developer API.

Table 2.2: Categorization of Pregel-like graph processing systems based on their core features.

System	**Distributed**	**Developer API**		**System Execution**				**Serializability**
		Programming Model	*Computation Model*	*Computation Model*	*Synchrony*	*Push/Pull*	*Global Barriers*	
Blogel [80]	✓	p-centric	BSP	AP	partial async[*]	push	✓	✗
CatchW [65]	✓	v-centric	BSP	BSP	sync	push	✓	✗
Giraph [1]	✓	v-centric	BSP	BSP	sync	push	✓	✗
Giraph++ [72] (hybrid mode)	✓	p-centric v-centric	BSP AP	AP AP	partial async[*] partial async[*]	push	✓	✗
Giraph async [38]	✓	v-centric	BSP/eAP[†]	eAP[†]	async	push	✓	✓
GiraphUC [38]	✓	v-centric	BSP/eAP[†]	**BAP**	async	push	**minimal**	✓
Giraphx [71]	✓	v-centric	BSP/AP	AP	async	push	✓	✓[‡]
GPS [63]	✓	v-centric	BSP	BSP	sync	push	✓	✗
GRACE [74]	✗	v-centric	AP	AP	async	push	✓	✗
GraphLab [50]/ Powergraph [34]	✓	v-centric	GAS	GAS	either	both	✓(sync), ✗(async)	✓
GraphX [78]	✓	v-centric	BSP/GAS	BSP/GAS	sync	both	✓	✗
LFGraph [41]	✓	v-centric	BSP	BSP	sync	pull	✓	✗
Mizan [45]	✓	v-centric	BSP	BSP	sync	push	✓	✗
MOCgraph [87]	✓	v-centric	BSP/AP	BSP/AP	either	push	✓	✗
PowerSwitch [75]	✓	v-centric	GAS	GAS	both	both	✓(sync), ✗(async)	✗
Pregel [54]	✓	v-centric	BSP	BSP	sync	push	✓	✗
Pregelix [20]	✓	v-centric	BSP	BSP	sync	push	✓	✗
Seraph [79]	✓	v-centric	GES[§]	GES[§]	sync	push	✓	✗
Signal/Collect [69]	✓	v-centric	GAS[¶]	GAS[¶]	either	both	✓(sync), ✗(async)	✗
X-Pregel [12]	✓	v-centric	BSP	BSP	sync	push	✓	✗

[*] Async within each partition/block of vertices but sync between partitions. Each worker machine can have multiple partitions.

[†] eAP is the enhanced AP model, which provides better algorithmic support than AP (Chapter 3.1.2).

[‡] Serializability is provided as part of specific user algorithms rather than as a generic and optional system feature (Chapter 2.2.2.4).

[§] GES is BSP with message generation separated out into an explicit `generate()` function (Chapter 2.3).

[¶] Signal is equivalent to scatter, collect is equivalent to gather and apply.

Table 2.3: Categorization of Pregel-like graph processing systems based on their additional features.

System	**Graph Partitioning**			**Optimizations**			**Graph Mutations**	**Multi-phase Algorithms**
	Cut Type	*Static Partitioning*	*Dynamic Migration*	*Message Combiner*	*Recv-side Scatter*	*Message Batching*		
Blogel [80]	edge	Graph Voroni Diagram, 2D partitioner	✗	receiver	✗	✓	✓	✓
CatchW [65]	edge	hash	✓	receiver	✗	✓	✗	✓
Giraph [1]	edge	hash	✗	receiver	✗	✓	✓	✓
Giraph++ [72] (hybrid mode)	edge	graph coarsening hash	✗	receiver	✗	✓	✓	✓ ✗
Giraph async [38]	edge	hash	✗	receiver	✗	✓	✓	✓
GiraphUC [38]	edge	hash	✗	receiver	✗	✓	✓	✓
Giraphx [71]	edge	hash, mesh	✗	receiver	✗	✓	✓	✗
GPS [63]	edge	hash	✓	receiver	✓	✓	partial	✓
GRACE [74, 76]	edge	METIS	✗	✗	✗	✓	✗	✓
GraphLab [50]/ Powergraph [34]	vertex	hash, oblivious, grid, PDS	✗	receiver	✗	✓(sync), ✗(async)	partial	✓
GraphX [78]	vertex	hash, 2D hash	✗	receiver	✗	✓	✗	✓
LFGraph [41]	edge	hash	✗	✗	✓	✓	✗	✓
Mizan [45]	edge	hash	✓	receiver	✗	✓	✓	✓
MOCgraph [87]	edge	hash	✗	receiver	✗	✓	✗	✓(sync), ✗(async)
PowerSwitch [75]	vertex	hash, oblivious, grid, PDS	✗	receiver	✗	✓(sync), ✗(async)	partial	✓
Pregel [54]	edge	hash	✗	both	✗	✓	✓	✓
Pregelix [20]	edge	hash	✗	sender	✗	✓	✓	✓
Seraph [79]	edge	hash	✗	✗	✗	✓	✓	✓
Signal/Collect [69]	edge	hash	✗	both	✗	✓(sync), ✗(async)	✓	✓
X-Pregel [12]	edge	hash	✓	sender	✓	✓	partial	✓

2.3 Pregel-like Graph Processing Systems

We now provide some additional details for a subset of the Pregel-like systems described in Chapter 2.2 and categorized in Tables 2.2 and 2.3.

CatchW is built on top of Apache Hama [3], which is a general BSP system that is not optimized for graph processing and does not support graph mutations. As described in Chapter 2.2.3.1, CatchW and Mizan are systems that focus on more sophisticated dynamic migration schemes, whereas GPS and X-Pregel offer simpler and more lightweight schemes. X-Pregel [12] is an implementation of Pregel using IBM's X10 programming language.

Giraph++ [72] primarily focuses on a graph-centric programming model but also considers a separate vertex-centric hybrid mode that implements a partially async AP model (Chapter 2.2.2). GRACE [74] is a single-machine shared memory system that is not distributed and does not support disk-based computation. GRACE implements the AP model through user customizable vertex scheduling and message selection, which can complicate the developer API. In contrast, GiraphUC preserves Giraph's simple BSP developer interface. GRACE also requires an immutable graph structure and so it does not support graph mutations.

GraphX [78] is built on the data parallel engine Spark [82] and presents a Resilient Distributed Graph (RDG) programming abstraction in which graphs are stored as tabular data and graph operations are implemented using distributed joins. GraphX supports both BSP and sync GAS (but not async GAS) and its primary goal is to provide more efficient graph processing for end-to-end data analytic pipelines implemented in Spark. Similarly, Pregelix [20] implements BSP graph processing in Hyracks [18], a shared-nothing dataflow engine, and stores graphs and messages as data tuples and uses joins to implement message passing.

The design of LFGraph [41] is based on the notion that hash partitioning is sufficient for static graph partitioning and that combiners are unnecessary. It is unique in that it is a pull-based BSP system where vertices store only their in-edges. In contrast, nearly all other BSP systems are push-based and have vertices store their out-edges.

MOCgraph [87] is a system that provides a combiner optimization, called message online computing (MOC), which immediately applies received messages to the recipient's vertex value instead of buffering them in a message store and applying them when the vertex is next executed. Similar to message combiners (Chapter 2.2.3.2), the combine operation is specified by a user-defined `onlineCompute()` function that must be commutative and associative. MOCgraph also considers an optimization for out-of-core (disk-based) computation, where each machine exchanges vertices between its own partitions such that hot vertices are co-located on the same partitions. Machines can then keep these hot partitions in-memory and, instead of frequently loading cold partitions into memory (which is costly), dump messages for cold partitions directly to disk and only occasionally load cold partitions into memory to apply these messages.

Seraph [79] is a vertex-centric BSP system that allows multiple algorithms to execute on a shared input graph. This is in contrast to existing systems, where executing multiple algorithms requires each algorithm to store its own copy of the input graph in memory, as such systems focus on executing only one algorithm at a time. Seraph's approach allows the input graph to be shared, which considerably lowers the memory costs of algorithms operating on the same input graph. It introduces a new GES model, where developers place message generation code inside a `generate()` function rather than in the per-vertex compute function. The GES model is otherwise identical to BSP.

Lastly, Signal/Collect [69] was one of the first systems to consider an explicit two-phase gather-scatter approach. This is different from systems like Pregel and Giraph, which have an explicit scatter (send) but implicit gather (receive). The scatter-gather model is essentially identical to GraphLab's GAS model, only with gather and apply occurring as a single phase.

2.4 Other Related Systems

Finally, there are several other systems related to graph processing but deviate from our core focus on specialized in-memory Pregel-like graph processing systems. These additional systems can be broadly categorized as iterative MapReduce systems, graph databases, single-machine disk-based systems, or Pregel-like graph processing on generic big data systems.

MapReduce [29] is an early programming model introduced for performing large-scale distributed data computations. However, it does not natively support iterative computation, which is required by graph processing. Consequently, performing graph processing on MapReduce incurs significant performance penalties: the input graph is shuffled between mappers and reducers on every iteration, which adds substantial communication overheads. Furthermore, the graph is written to disk at the end of each MapReduce iteration and streamed back into memory for the next iteration, resulting in unnecessary and costly disk I/O. HaLoop [21], iMapReduce [83], PrIter [84], Stratophere [10], Surfer [23], and Twister [31] are all systems developed to both reduce these performance overheads and improve the developer interface for iterative MapReduce. While these systems outperform regular MapReduce systems like Hadoop [2], their performance on graph processing tasks remains worse than specialized Pregel-like graph processing systems. For example, Stratosphere is up to two times slower than Giraph 0.2 [10], a version of Giraph that is over two years older than the current and substantially more performant Giraph 1.1.0 [1].

Graph databases are systems that persistently store databases in the format of a graph. In contrast to iterative MapReduce and Pregel-like systems, graph databases focus on providing persistence graph storage and support for online queries on the stored graph. Consequently, existing graph databases have either little or no support for the sophisticated offline graph analytics possible in specialized Pregel-like systems. For example, the popular graph database Titan [7] features a graph analytics engine Faunus [61], which uses MapReduce instead of a Pregel-like approach. Other popular graph databases include OrientDB [6] and Neo4j [4].

In addition to distributed in-memory Pregel-like systems, there are currently two other important avenues that researchers are investigating.

The first is single-machine disk-based Pregel-like systems, which spill large input graphs out to disk. GraphChi [48] was one of the first vertex-centric systems to consider this approach and introduced a parallel sliding windows technique to minimize non-sequential disk I/O. X-Stream [62], in contrast, uses an edge-centric programming model (Chapter 2.2.1) to avoid random I/O. TurboGraph [40] further improves on GraphChi by using techniques that exploit the I/O parallelism of SSDs. More recent systems include FlashGraph [86], which considers techniques for arrays of SSDs, and VENUS [24], which introduces a more efficient model of storing and accessing graph data on disk as well as caching strategies to further improve performance.

The second avenue is supporting Pregel-like graph processing on more generic big data systems, such as relational databases. This direction is motivated by the need for better integration with existing big data tools, as well as the fact that a large portion of the data that developers want to analyze are already stored in a relational format: using specialized Pregel-like systems would first require converting this data into a suitable graph format before analytics can occur. As described in Chapter 2.3, GraphX [78] and Pregelix [20] are designed with this in mind: they both store graphs as tabular data, which easily integrates with the tabular format used by the other tools of the data engines on which they are built. Consequently, they are more generic than, for example, Giraph and GraphLab. Similarly, Trinity [66] uses a globally addressable in-memory key-value store, where data is partitioned across workers by their keys rather than by vertex or edge IDs, and it supports both online queries and offline graph analytics. Aster Graph Analytics [68], GRAIL [32], and Vertexica [42] address Pregel-like graph processing on relational databases by mapping a subset of graph analytic tasks to relational operations.

Chapter 3

Giraph Unchained: Barrierless Asynchronous Parallel Execution

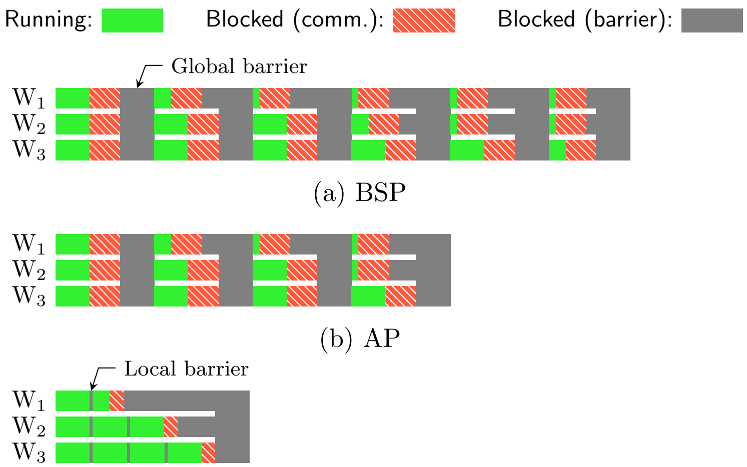
We begin this chapter by motivating BAP through a detailed discussion of the shortcomings of the BSP, AP, and GAS models. We then introduce our BAP model and its implementation, GiraphUC, in Chapters 3.2 and 3.3. We provide an experimental evaluation in Chapter 3.4 and conclude with a summary in Chapter 3.5.

3.1 Motivation

3.1.1 Performance

In BSP, global barriers ensure that all messages sent in one superstep are delivered before the start of the next superstep, thus resolving implicit data dependencies encoded in messages. However, the synchronous execution enforced by these global barriers causes BSP to suffer from major performance limitations: stale messages, large communication overheads, and high synchronization costs due to stragglers.

To illustrate these limitations concretely, Figures 3.1a and 3.1b visualize the BSP and AP executions of our WCC example (Chapter 2.1) with explicit global barriers and with time flowing horizontally from left to right. The green regions indicate computation while the red striped and gray regions indicate that a worker is blocked on communication or on a barrier, respectively. For simplicity, assume that the communication overheads and global barrier processing times are constant.



(a) BSP

(b) AP

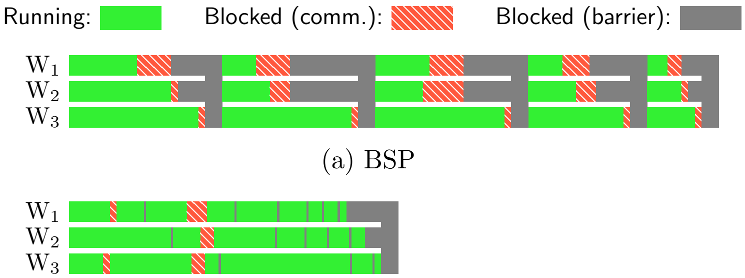
(c) BAP

Figure 3.1: Computation times for the WCC example under different computation models.

Stale messages. As described in Chapter 2.1.2, reducing stale messages allows the AP execution of WCC to finish in fewer supersteps than the BSP execution, translating to shorter computation time. In general, allowing vertices to see more recent (less stale) data to update their per-vertex parameters enables faster convergence and shorter computation times, resulting in better performance [50]. Our proposed BAP model preserves these advantages of the AP model by also reducing message staleness without using distributed locking (Chapter 3.2).

Communication overheads. Since BSP and AP execute only one superstep between global barriers, there is usually insufficient computation work to adequately overlap with and mask communication overheads. For example, in Figures 3.1a and 3.1b, workers spend a large portion of their time blocked on communication. Furthermore, for AP, the communication overheads can outweigh performance gains achieved by reducing message staleness. Figure 3.1c illustrates how our proposed BAP model resolves this by minimizing the use of global barriers: each worker can perform multiple *logical* supersteps (separated by inexpensive *local* barriers) without global barriers (Chapter 3.2), which greatly improves the overlap between computation and communication.

Stragglers and synchronization costs. Stragglers are the slowest workers in a computation. They are caused by a variety of factors, some as simple as unbalanced hardware resources and others that are more complex. For example, the power-law degree distribution of natural graphs used in computations can result in substantial computation and communication load for a handful of workers due to the extremely high degrees of a



(a) BSP

(b) BAP

Figure 3.2: WCC computation times based on real executions of 16 workers on `TW` (Table 3.1).

small number of vertices [70]. In algorithms like PageRank, some regions of the graph may converge much slower than the rest of the graph, leading to a few very slow workers.

The use of global barriers then gives rise to the straggler problem: global synchronization forces all fast workers to block and wait for the stragglers. Consequently, fast workers spend large portions of their time waiting for stragglers rather than performing useful computation. Hence, global barriers carry a significant synchronization cost. Furthermore, because BSP and AP both use global barriers frequently, these synchronization costs are further multiplied by the number of supersteps executed. On graphs with very large diameters, algorithms like WCC can require thousands of supersteps, incurring substantial overheads.

As an example, consider Figure 3.2, which is based on real executions of a large real-world graph. In the BSP execution (Figure 3.2a), W_3 is the straggler that forces W_1 and W_2 to block on every superstep. This increases the overall computation time and prevents W_1 and W_2 from making progress between global barriers. The BAP model (Figure 3.2b) significantly lowers these synchronization overheads by minimizing the use of global barriers, which allows W_1 and W_2 to perform multiple iterations without waiting for W_3. Furthermore, W_1 and W_2 are able to compute further ahead and propagate much newer data to W_3, enabling W_3 to finish in less time under the BAP model.

3.1.2 Algorithmic Support

The AP model supports BSP algorithms that perform accumulative updates, such as WCC (Chapter 3.4.2.2), where a vertex does not need all messages from all neighbours to perform its computation (Theorem 1).

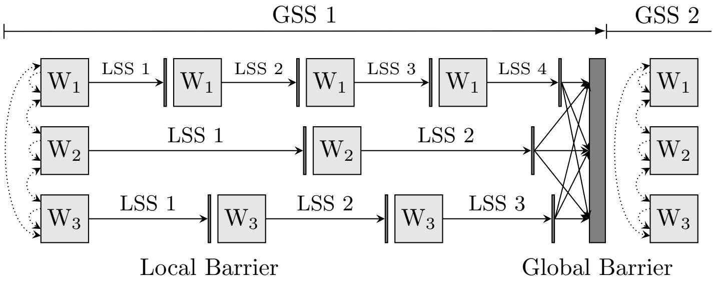


Figure 3.3: The BAP model, with two global supersteps and three workers. GSS stands for global superstep, while LSS stands for logical superstep.

Theorem 1. *The AP and BAP models correctly execute single-phase BSP algorithms in which vertices do not need all messages from all neighbours.*

Proof Sketch. (See Appendix A for full proof.) A key property of single-phase BSP algorithms is that (1) the computation logic is the same in each superstep. Consequently, it does not matter *when* a message is seen, because it will be processed in the same way. If vertices do not need all messages from all neighbours, then (2) the compute function can handle any number of messages in each superstep. Intuitively, if every vertex executes with the same logic, can have differing number of edges, and does not always receive messages from all neighbours, then it must be able to process an arbitrary number of messages.

Thus, correctness depends only on ensuring that every message from a vertex v to a vertex u is seen exactly once. Since both the AP and BAP model change only *when* messages are seen and not *whether* they are seen, they both satisfy this condition. For example, AP's relaxed isolation means that messages may be seen one superstep earlier. □

However, the AP model cannot handle BSP algorithms where a vertex must have all messages from all neighbours nor algorithms with multiple computation phases. In contrast, the BAP model supports both types of algorithms. Furthermore, as we show in the next section, we can add BAP's support of these two types of algorithms back into AP to get the *enhanced AP* (eAP) model.

3.2 The BAP Model

Our barrierless asynchronous parallel (BAP) model offers an efficient asynchronous execution mode by reducing both the staleness of messages and frequency of global barriers.

As discussed in Chapter 3.1.1, global barriers limit performance in both the BSP and AP models. The BAP model avoids global barriers by using *local barriers* that separate *logical supersteps*. Unlike global barriers, local barriers do not require global coordination: they are local to each worker and are used only as a pausing point to perform tasks like graph mutations and to decide whether a global barrier is necessary. Since local barriers are internal to the system, they occur automatically and are transparent to developers.

A logical superstep is logically equivalent to a regular BSP superstep in that both execute vertices exactly once and are numbered with strictly increasing values. However, unlike BSP supersteps, logical supersteps are not globally coordinated and so different workers can execute a different number of logical supersteps. We use the term *global supersteps* to refer to collections of logical supersteps that are separated by global barriers. Figure 3.3 illustrates two global supersteps (GSS 1 and GSS 2) separated by a global barrier. In the first global superstep, worker 1 executes four logical supersteps, while workers 2 and 3 execute two and three logical supersteps respectively. In contrast, BSP and AP have exactly one logical superstep per global superstep.

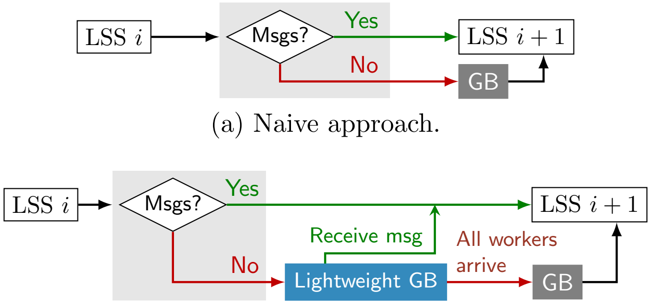
Local barriers and logical supersteps enable fast workers to continue execution instead of blocking, which minimizes communication and synchronization overheads and mitigates the straggler problem (Chapter 3.1.1). Logical supersteps are thus much cheaper than BSP supersteps as they avoid synchronization costs. Local barriers are also much faster than the processing times of global barriers alone (i.e., excluding synchronization costs), since they do not require global communication. Hence, per-superstep overheads are substantially smaller in the BAP model, which results in significantly better performance.

Finally, as in the AP model, the BAP model reduces message staleness by allowing vertices to immediately see local and remote messages that they have received. In Figure 3.3, dotted arrows represent messages received and seen/processed in the same logical superstep, while solid arrows indicate messages that are not processed until the next logical superstep. For clarity, we omit dotted arrows between every worker box but note that they do exist.

Next, we present details about local barriers and algorithmic support in the BAP model.

3.2.1 Local Barriers

For simplicity, we first focus on algorithms with a single computation phase. Algorithms with multiple computation phases are discussed in Chapter 3.2.3.



(a) Naive approach.

(b) Improved approach.

Figure 3.4: Simplified comparison of worker control flows for the two approaches to local barriers. LSS stands for logical superstep and GB for global barrier.

3.2.1.1 Naive Approach

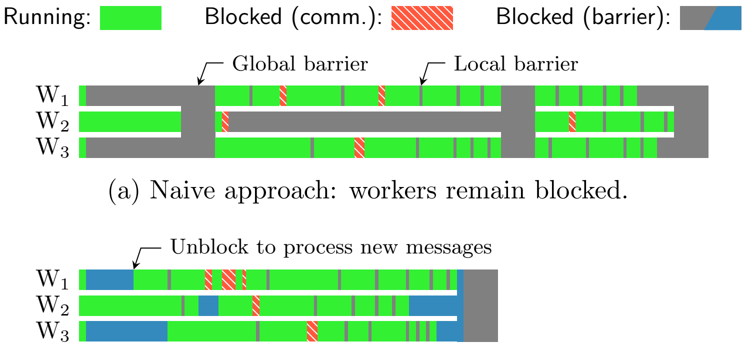
The combination of relaxed message isolation and local barriers allow workers to compute without frequently blocking and waiting for other workers. However, this can pose a problem for termination as both the BSP and AP models use the master to check the termination conditions at the global barrier following the end of every BSP superstep.

To resolve this, we use a two step termination check. The first step occurs locally at a local barrier, while the second step occurs globally at the master. Specifically, at a local barrier, a worker independently decides to block on a global barrier if there are no more local or remote messages to process since this indicates there is no more work to be done (Figure 3.4a). We do not need to check if all local vertices are inactive since any pending messages will reactivate vertices. After all workers arrive at the global barrier, the master executes the second step, which is simply the regular BSP termination check: terminate if all vertices are inactive and there are no more unprocessed messages.

Since remote messages can arrive asynchronously, we must count them carefully to ensure that received but unprocessed messages are properly reported to the master as unprocessed messages (Chapter 3.3.3). This prevents the master from erroneously terminating the algorithm.

3.2.1.2 Improved Approach

The above approach is naive because it does not take into account the arrival of remote messages after a worker decides to block on a global barrier. That is, newly received remote messages are not processed until the next *global* superstep. This negatively impacts



(a) Naive approach: workers remain blocked.

(b) Improved approach: workers unblock to process new messages. Blue regions indicate lightweight global barrier.

Figure 3.5: Performance comparison between the naive vs. improved approach, based on SSSP executions of 16 workers on `TW` (Table 3.1).

performance as workers can remain blocked for a long time, especially in the presence of stragglers, and it also results in more frequent global barriers since the inability to unblock from a barrier means that all workers eventually stop generating messages.

For example, the single source shortest path (SSSP) algorithm begins with only one active vertex (the source), so the number of messages sent between vertices increases, peaks and then decreases with time (Chapter 3.4.2.1). Figure 3.5a shows an SSSP execution using the naive approach, where W_1 and W_3 initially block on a global barrier as they have little work to do. Even if messages from W_2 arrive, W_1 and W_3 will remain blocked. In the second global superstep, W_2 remains blocked for a very long time due to W_1 and W_3 being able to execute many logical supersteps before running out of work to do. As a result, a large portion of time is spent blocked on global barriers.

However, if we allow workers to unblock and process new messages with additional logical supersteps, we can greatly reduce the unnecessary blocking and shorten the total computation time (Figure 3.5b). The improved approach does precisely this: we insert a lightweight global barrier, before the existing (BSP) global barrier, that allows workers to unblock upon receiving a new message (Figure 3.4b). This additional global barrier is lightweight because it is cheaper to block and unblock from compared to the (BSP) global barrier (Chapter 3.3.3). Unblocking under the above condition is also efficient because messages arrive in batches (Chapter 3.3.2), so there is always sufficient new work to do.

Additionally, with this improved approach, if each worker W_i waits for all its sent messages to be delivered (acknowledged) before blocking, the recipient workers will unblock before W_i can block. This means that all workers arrive at the lightweight global barrier (and proceed to the (BSP) global barrier) only when there are no messages among any workers. This allows algorithms with a single computation phase to be completed in exactly one global superstep.

Hence, local barriers ensure that algorithms are executed using the minimal number of global supersteps, which minimizes communication and synchronization overheads. Furthermore, the two step termination check is more efficient and scalable than the distributed consensus algorithm used by GraphLab async, as our experimental results will show (Chapter 3.4.3). Finally, unlike GraphLab and its GAS model, BAP fully supports graph mutations by having workers resolve pending mutations during a local barrier (Chapter 3.3.4).

3.2.2 Algorithmic Support

The BAP model, like the AP model, supports single-phase algorithms that do not need all messages from all neighbours (Theorem 1). Theorem 2 shows how the BAP model also supports algorithms where vertices do require all messages from all neighbours. This theorem also improves the algorithmic support of the AP model, to give the eAP model.

Theorem 2. *Given a message store that is initially filled with valid messages, retains old messages, and overwrites old messages with new messages, the BAP model correctly executes single-phase BSP algorithms in which vertices need all messages from all neighbours.*

Proof Sketch. (See Appendix A for full proof.) Like in Theorem 1's proof sketch, we again have property (1), so when a message is seen is unimportant. Since vertices need all messages from all (in-edge) neighbours, we also have that (2) an old message m from vertex v to u can be overwritten by a new message m' from v to u. The intuition is that since every vertex needs messages from all its in-edge neighbours, it must also send a message to each of its out-edge neighbours. This means a newer message contains a more recent state of a neighbour, which can safely overwrite the old message that now contains a stale state.

Correctness requires that every vertex u sees exactly one message from each of its in-edge neighbours in each (logical) superstep. That is, u must see exactly $N = \deg^{-}(u)$ messages. The message store described in the theorem ensures that each u starts with, and (by retaining old messages) will always have, exactly N messages. Property (2) allows new incoming messages to overwrite corresponding old messages, again ensuring N messages. Thus, independent of the (logical) superstep of execution, u always sees N messages, so both the AP and BAP models guarantee correctness. □

Per Theorem 2, the message store must be initially filled with messages. This is achieved in the BAP model by adding a global barrier after the very first logical superstep of the algorithm, when messages are sent for the first time.

3.2.3 Multiple Computation Phases

Algorithms with multiple computation phases are computations composed of multiple tasks, where each task has different compute logic. Therefore, computation phases require global coordination for correctness. To do so otherwise requires rewriting the algorithm such that it can no longer be programmed for BSP, which negatively impacts usability.

For example, in DMST (Chapter 3.4.2.3), the phase where vertices find a minimum weight out-edge occurs after the phase in which vertices add and remove edges. If these two phases are not separated by a global barrier, the results will be incorrect as some vertices will not have completed their mutations yet. Hence, the BAP model must use global barriers to separate different computation phases. However, we can continue to use local barriers and logical supersteps within each computation phase, which allows each phase to complete in a single global superstep (Chapter 3.2.1.2).

In addition to computation phases that have multiple supersteps, many multi-phase algorithms have single-superstep phases (phases that are only one BSP superstep). These are typically used to send messages to be processed in the next phase. Even with global barriers separating computation phases, relaxed message isolation will cause vertices to see these messages in the incorrect phase. In other words, messages for different phases will be incorrectly mixed together. We handle these heterogeneous messages by considering all the possible scenarios, as proved in Theorem 3.

Theorem 3. *If every message is tagged with a Boolean at the API level and two message stores are maintained at the system level, then the BAP model supports BSP algorithms with multiple computation phases.*

Proof. Since the BAP model executes only algorithms written for the BSP model, it suffices to consider multi-phase algorithms implemented for BSP. For such algorithms, there are only three types of messages: (1) a message sent in the previous phase $k-1$ to be processed in the current phase k; (2) a message sent in phase k to be processed in this same phase k; and (3) a message sent in phase k to be processed in the next phase $k+1$. Other scenarios, in which a message is sent in phase m to be processed in phase n where $(n-m) > 1$, are not possible due to a property of BSP: any message sent in one superstep is visible only in the next superstep, so messages not processed in the next superstep are lost. If

$(n - m) > 1$, then phases n and m are more than one superstep apart and so n cannot see the messages sent by m. Since BAP separates computation phases with global barriers, it inherits this property from the BSP model.

For the two message stores, we use one store (call it MS_C) to hold messages for the current phase and the other store (call it MS_N) to hold messages for the next phase. When a new computation phase occurs, the MS_N of the previous phase becomes the MS_C of the current phase, while the new MS_N becomes empty. We use a Boolean to indicate whether a message, sent in the current phase, is to be processed in the next computation phase. Since all messages are tagged with this Boolean, we can determine which store (MS_C or MS_N) to store received messages into.

Thus, all three scenarios are covered: if (1), the message was placed in MS_N during phase $k - 1$, which becomes MS_C in phase k and so the message is made visible in the correct phase; if (2), the message is placed in MS_C of phase k, which is immediately visible in phase k; finally, if (3), the message is placed in MS_N of phase k, which will become the MS_C of phase $k + 1$. □

Theorem 3 implicitly requires knowledge of when a new phase starts in order to know when to make MS_N the new MS_C. In BAP, the start and end of computation phases can be inferred locally by each worker since each phase completes in one global superstep. That is, the start of a phase is simply the start of a global superstep, while the end of a phase is determined by checking if any more local or remote messages are available for the current phase. Messages for the next phase are ignored, as they cannot be processed yet. Hence, the BAP model detects phase transitions without modifications to the developer API.

Theorem 3 is also applicable to the AP model, thus enhancing it to also support multi-phase algorithms. However, unlike BAP, the eAP model is unable to infer the start and end of computation phases, so it requires an additional API call for algorithms to notify the system of a computation phase change (Chapter 3.3.5.1).

Chapter 3.3.5 describes an implementation that provides message tagging without introducing network overheads. Hence, the BAP model efficiently supports multi-phase algorithms while preserving the BSP developer interface.

3.3 GiraphUC

We now describe GiraphUC, our implementation of the BAP model in Giraph. We use Giraph because it is a popular and performant push-based distributed graph processing system. For example, Giraph is used by Facebook in their production systems [25]. We first provide background on Giraph, before discussing the modifications done to implement the eAP model and then the BAP model.

3.3.1 Giraph Background

Giraph is an open source system that features receiver-side message combining (to reduce memory usage and computation time), blocking aggregators (for global coordination or counters), and `master.compute()` (for serial computations at the master). It supports multithreading by allowing each worker to execute with multiple compute threads. Giraph reads input graphs from, and writes output to, the Hadoop Distributed File System (HDFS).

Giraph partitions input graphs using hash partitioning and assigns multiple graph partitions to each worker. During each superstep, a worker creates a pool of compute threads and pairs available threads with uncomputed partitions. This allows multiple partitions to be executed in parallel. Between supersteps, workers execute with a single thread to perform serial tasks like resolving mutations and blocking on global barriers. Communication threads are always running concurrently in the background to handle incoming and outgoing requests.

Each worker maintains its own message store to hold all incoming messages. To reduce contention on the store and efficiently utilize network resources, each compute thread has a message buffer cache to batch all outgoing messages. Namely, messages created by vertex computations are serialized to this cache. After the cache is full or the partition has been computed, the messages are flushed to the local message store (for local messages) or sent off to the network (for remote messages). Use of this cache is the primary reason Giraph does not perform sender-side message combining: there are generally insufficient messages to combine before the cache is flushed, so combining adds more overheads than benefits [53].

To implement BSP, each worker maintains two message stores: one holding the messages from the previous superstep and another holding messages from the current superstep. Computations see only messages in the former store, while messages in the latter store become available in the next superstep. At the end of each superstep, workers wait for all outgoing messages to be delivered before blocking on a global barrier. Global synchronization is coordinated by the master using Apache ZooKeeper.

3.3.2 Giraph Async

We first describe Giraph async, our implementation of the enhanced AP (eAP) model that has support for additional types of algorithms (Chapter 3.1.2). Just as the AP model is the starting point for the BAP model, implementing Giraph async is the first step towards implementing GiraphUC.

Note that GRACE and Giraph++'s hybrid mode do not implement the eAP model. Specifically, GRACE is a single-machine shared memory system and does not support graph mutations. Giraph++'s hybrid mode is distributed but relaxes message isolation only for local messages and it does not support multi-phase algorithms. In contrast, Giraph async relaxes message isolation for both local and remote messages and fully supports graph mutations and multi-phase algorithms.

Giraph async provides relaxed message isolation by using a single message store that holds messages from both the previous and current supersteps. For GiraphUC, this would be the previous and current logical, rather than BSP, supersteps.

We allow outgoing local messages to skip the message buffer cache and go directly to the message store. This minimizes staleness since a local message becomes immediately visible after it is sent. While this slightly increases contention due to more frequent concurrent accesses to a shared message store, there is a net gain in performance from the reduction in message staleness.

For outgoing remote messages, we continue to use the message buffer cache, as message batching dramatically improves network performance. Specifically, given the network latencies between machines, removing message batching to minimize staleness only degrades performance. This contrasts with GraphLab async, whose pull-based approach hinders the ability to batch communication.

When computing a vertex, messages that have arrived for that vertex are removed from the message store, while messages that arrive after are seen in the next (logical) superstep. For algorithms in which vertices require all messages from all neighbours (Theorem 2), the messages for a vertex are retrieved but not removed, since the message store must retain old messages. To allow the message store to identify which old messages to overwrite, we transparently tag each message with the sender's vertex ID, without modification to the developer API. Chapter 3.3.5 describes how we support algorithms with multiple computation phases.

In Giraph, graph mutations are performed after a global barrier. Since Giraph async retains these global barriers, it naturally supports mutations in the same way. Chapter 3.3.4 describes how mutations are supported in GiraphUC.

3.3.3 Adding Local Barriers

To implement GiraphUC, we add local barriers to Giraph async. We implement local barriers following the improved approach described in Chapter 3.2.1.2.

In the first step of the termination check, workers check whether their message store is empty or, if messages are overwritten instead of removed, whether any messages were overwritten. In the second step, the master checks if all vertices are inactive and the number of unprocessed messages is zero, based on statistics that each worker records with ZooKeeper. In BSP, the number of unprocessed messages is simply the number of sent messages. In BAP, however, this number fails to capture the fact that remote messages can arrive at any time and that they can be received and processed in the same global superstep (which consists of multiple logical supersteps). Hence, we assign each worker a byte counter that increases for sent remote messages, decreases for received and processed remote messages, and is reset at the start of every global (but not logical) superstep. Each worker records the counter's value with ZooKeeper before blocking at a global barrier. This ensures that any received but unprocessed messages are recorded as unprocessed messages. By summing together the workers' counters, the master correctly determines the presence or absence of unprocessed messages.

Finally, the lightweight global barrier is also coordinated by the master via ZooKeeper but, unlike the (BSP) global barrier, does not require workers to record any statistics with ZooKeeper before blocking. This allows workers to unblock quickly without needing to erase recorded statistics. Also, as described in Chapter 3.2.1.2, workers wait for all sent messages to be acknowledged before blocking on the lightweight barrier, which ensures that each computation phase completes in a single global superstep.

3.3.4 Graph Mutations

GiraphUC, unlike GraphLab, fully supports graph mutations. Mutation requests are sent as asynchronous messages to the worker that owns the vertices or edges being modified and the requests are buffered by that worker upon receipt.

In Giraph, and hence GiraphUC, a vertex is owned solely by one partition, while an edge belongs only to its source vertex (an undirected edge is represented by two directed edges). That is, although edges can cross partition (worker) boundaries, they will always belong to one partition (worker). Hence, vertex and edge mutations are both operations local to a single worker. Since mutations touch data structures shared between partitions, they can be conveniently and safely resolved during a local barrier, when no compute threads are executing.

Since mutation requests and regular vertex messages are both asynchronous, they may arrive out of order. However, this is not a problem as all messages are buffered in the recipient worker's message store. If the messages are for a new vertex, they will remain in the store until the vertex is added and retrieves said messages from the store by itself. If the messages are for a deleted vertex, they will be properly purged, which is identical to the behaviour in BSP (recall that BSP uses rotating message stores). More generally, if a BSP algorithm that performs vertex or edge mutations executes correctly in BSP, then it will also execute correctly in BAP. We have additionally verified correctness experimentally, using both algorithms that perform edge mutations, such as DMST (Chapter 3.4.2.3), and algorithms that perform vertex mutations, such as k-core [59, 5]. Even non-mutation algorithms like SSSP (Chapter 3.4.2.1) can perform vertex additions in Giraph: if an input graph does not explicitly list a reachable vertex, it gets added via vertex mutation when first encountered.

3.3.5 Multiple Computation Phases

As proved for Theorem 3, only one simple change to the developer API is necessary to support multi-phase algorithms: all messages must be tagged with a Boolean that indicates whether the message is to be processed in the next computation phase. This addition does not impede usability since the Boolean is straightforward to set: `true` if the phase sending the message is unable to process such a message and `false` otherwise. For example, this change adds only 4 lines of code to the existing 1300 lines for DMST (Chapter 3.4.2.3).

To avoid the network overheads of sending a Boolean with every message, we note that messages in Giraph are always sent together with a destination partition ID, which is used by the recipient to determine the destination graph partition of each message. Hence, we encode the Boolean into the integer partition ID: messages for the current phase have a positive partition ID, while messages for the next phase have a negative partition ID. The sign of the ID denotes the message store, MS_C or MS_N (Theorem 3), that the message should be placed into.

Finally, as per Chapter 3.2.3, the start and end of computation phases are, respectively, inferred by the start of a global superstep and the absence of messages for the current phase. The per-worker byte counters (Chapter 3.3.3) continue to track messages for both the current phase and the next phase. This ensures that the master, in the second step of the termination check, knows whether there are more computation phases (global supersteps) to execute.

3.3.5.1 Giraph Async

Giraph async uses the same Boolean tagging technique to support multi-phase algorithms. However, unlike GiraphUC, Giraph async cannot infer the start and end of computation phases, so algorithms must notify the system when a new computation phase begins (Chapter 3.2.3). Giraph async requires a parameterless notification call to be added either in `master.compute()`, where algorithms typically manage internal phase transition logic, or in the vertex compute function. In the former case, the master notifies all workers before the start of the next superstep. This allows workers to discern between phases and know when to exchange the MS_C and MS_N message stores (Theorem 3).

3.3.6 Aggregators and Combiners

Since Giraph is based on the BSP model, aggregators are blocking by default. That is, aggregator values can be obtained only after a global barrier. To avoid global barriers, GiraphUC supports aggregators that do not require global coordination. For example, algorithms that terminate based on some accuracy threshold use an aggregator to track the number of active vertices and terminate when the aggregator's value is zero. This works in GiraphUC without change since each worker can use a local aggregator that tracks its number of active vertices, aggregate the value locally on each logical superstep, and block on a global barrier when the local aggregator's value is zero. This then allows the master to terminate the computation.

Finally, like Giraph, GiraphUC supports receiver-side message combining and does not perform sender-side message combining as it also uses the message buffer cache for outgoing remote messages (Chapter 3.3.1).

3.3.7 Fault Tolerance

Fault tolerance in GiraphUC is achieved using Giraph's existing checkpointing and failure recovery mechanisms. Just as in Giraph, all vertices, edges, and message stores are serialized during checkpointing and deserialized during recovery. In the case of algorithms with multiple computation phases, checkpointing can be performed at the global barriers

Table 3.1: Directed datasets. Parentheses indicate values for the undirected versions used by DMST.

Graph	$\|V\|$	$\|E\|$	**Average Degree**	**Max In/Outdegree**	**Harmonic Diameter**
USA-road-d **(US)**	23.9M	57.7M (57.7M)	2.4 (2.4)	9 / 9 (9)	1897 ± 7.5
arabic-2005 **(AR)**	22.7M	639M (1.11B)	28 (49)	575K / 9.9K (575K)	22.39 ± 0.197
twitter-2010 **(TW)**	41.6M	1.46B (2.40B)	35 (58)	770K / 2.9M (2.9M)	5.29 ± 0.016
uk-2007-05 **(UK)**	105M	3.73B (6.62B)	35 (63)	975K / 15K (975K)	22.78 ± 0.238

that separate the computation phases. For more fine-grained checkpointing, or in the case of algorithms with only a single computation phase, checkpointing can be performed at regular time intervals instead. After each time interval, workers independently designate the next local barrier as a global barrier to enable a synchronous checkpoint.

3.4 Experimental Evaluation

We compare GiraphUC to synchronous Giraph (Giraph sync), Giraph async, GraphLab sync, and GraphLab async. We use these systems as both Giraph and GraphLab are widely used in academia and industry and are performant open source distributed systems [39]. While Giraph sync and GraphLab sync capture the performance of synchronous systems (BSP and async GAS, respectively), Giraph async is a performant implementation of the eAP model (Chapter 3.3.2) and GraphLab async is a state-of-the-art pull-based asynchronous system (async GAS).

We exclude GRACE and Giraph++'s hybrid mode, which both implement AP, because Giraph async is a more performant and scalable implementation of AP that also provides better algorithmic support (Chapter 3.3.2). Specifically, Giraph async is distributed, whereas GRACE is single-machine, and it is implemented on top of Giraph 1.1.0, which significantly outperforms the much older Giraph 0.1 on which Giraph++ is implemented. Giraph async also supports DMST, a multi-phase mutations algorithm, whereas GRACE and Giraph++'s hybrid mode do not. We also exclude systems like GPS, Mizan, and GraphX (Chapter 2.3) as they are less performant than Giraph and GraphLab [39, 78].

3.4.1 Experimental Setup

To test performance at scale, we use 64 EC2 r3.xlarge instances, each with four vCPUs and 30.5GB of memory. All machines run Ubuntu 12.04.1 with Linux kernel 3.2.0-70-virtual, Hadoop 1.0.4, and jdk1.7.0_65. We use Giraph 1.1.0-RC0 from June 2014, which is also the version that GiraphUC and Giraph async are implemented on, and the version of GraphLab 2.2 released in October 2014.

As scalability is a key focus, we evaluate all systems with large real-world datasets[1],[2][14, 16, 15]. We store all datasets as regular text files on HDFS and load them into all systems using the default random hash partitioning.

Table 3.1 lists the four graphs we test: US is a road network graph, TW is a social network graph, and AR and UK are both web graphs. Table 3.1 also details several properties for each graph. $|V|$ and $|E|$ denote the number of vertices and directed edges, while the average degree gives a sense of how large $|E|$ is relative to $|V|$. The maximum indegree or outdegree provides a sense of how skewed the graph's degree distribution is, while the harmonic diameter indicates how widely spread out the graph is [11, 17].

In particular, the social and web graphs all have very large maximum degrees since they follow a power-law degree distribution. Their small diameters also indicate tight graphs: TW, being a social graph, exhibits the "six degrees of separation" phenomenon, while the web graphs have larger diameters. In contrast, US has very small average and maximum degrees but a very large diameter. Intuitively, this is because cities (vertices) do not have direct roads (edges) to millions of other cities. Instead, most cities are connected by paths that pass through other cities, which means that road networks tend to sprawl out very widely—for example, US is spread across North America. These real-world characteristics can affect performance in different ways: high degree skews can cause performance bottlenecks at a handful of workers, leading to stragglers, while large diameters can result in slow convergence or cause algorithms to require a large number of supersteps to reach termination.

3.4.2 Algorithms

In our evaluation, we consider four different algorithms: SSSP, WCC, DMST, and PageRank. These four algorithms can be categorized in three different ways: compute boundedness, network boundedness, and accuracy requirements. PageRank is an algorithm that is computationally light, meaning it is proportionally more network bound, and it has a notion of accuracy. SSSP and WCC are also computationally light but do not have a notion of accuracy as their solutions are exact. Both are also network bound, with WCC requiring more communication than SSSP, and, unlike PageRank, the amount of communication in

[1] http://www.dis.uniroma1.it/challenge9/download.shtml

[2] http://law.di.unimi.it/datasets.php

SSSP and WCC also varies over time. Finally, DMST also provides exact solutions but it is computationally heavy and therefore more compute bound than network bound. Hence, each algorithm stresses the systems in a different way, providing insight into each system's performance characteristics.

The BSP implementations of SSSP, WCC, and PageRank work without modification on all systems. DMST is a multi-phase mutations algorithm that, while unsupported by GraphLab, can run on Giraph async and GiraphUC via a simple modification (Chapter 3.3.5). We next describe each algorithm in more detail.

3.4.2.1 SSSP

Single-source shortest path (SSSP) finds the shortest path between a source vertex and all other vertices in its connected component. We use the parallel variant of the Bellman-Ford algorithm [27] (Algorithm 1). Each vertex initializes its distance (vertex value) to ∞, while the source vertex sets its distance to 0. Vertices update their distance using the minimum distance received from their neighbours and propagate any newly discovered minimum distance to all neighbours. We use unit edge weights and the same source vertex to ensure that all systems perform the same amount of work.

Algorithm 1 SSSP pseudocode.

```
procedure COMPUTE(vertex, incoming messages)
    if superstep == 0 then
        vertex.setValue($\infty$)
        if vertex.getID() == $sourceVertex$ then
            $d_{min} \leftarrow 0$
    else
        $d_{min} \leftarrow$ minimum of all message values
    if $d_{min} <$ vertex.getValue() then
        vertex.setValue($d_{min}$)
        for all outgoing edges $e = (\text{vertex}, v)$ do
            Send $d_{min} + e$.getWeight() to $v$
    voteToHalt()
```

3.4.2.2 WCC

Weakly connected components (WCC) is an algorithm that finds the maximal weakly connected components of a graph. A component is weakly connected if all constituent vertices are mutually reachable when ignoring edge directions. We use the HCC algorithm [44], which starts with all vertices initially active (Algorithm 2). Each vertex initializes its component ID (vertex value) to its vertex ID. When a smaller component ID is received, the vertex updates its vertex value to that ID and propagates the ID to its neighbours. We correct GraphLab's WCC implementation so that it executes correctly in GraphLab async.

Algorithm 2 WCC pseudocode.

```
procedure COMPUTE(vertex, incoming messages)
    if superstep == 0 then
        vertex.setValue(vertex.getID())
    $compID_{min}$ ← minimum of all message values
    if $compID_{min}$ < vertex.getValue() then
        vertex.setValue($compID_{min}$)
        Send $compID_{min}$ to vertex's outgoing neighbours
    voteToHalt()
```

3.4.2.3 DMST

Distributed minimum spanning tree (DMST) finds the minimum spanning tree (MST) of an undirected, weighted graph. For unconnected graphs, DMST gives the minimum spanning forest, a union of MSTs. We use the parallel Boruvka algorithm [26, 64] and undirected versions of our datasets weighted with distinct random edge weights. We omit the pseudocode for DMST due to its complexity.

The algorithm has four different computation phases. In phase one, each vertex selects a minimum weight out-edge. In phase two, each vertex u uses its selected out-edge and the pointer-jumping algorithm [26] to find its supervertex, a vertex that represents the connected component to which u belongs. Phase two requires multiple supersteps to complete and is coordinated using summation aggregators. In phase three, vertices perform edge cleaning by deleting out-edges to neighbours with the same supervertex and modifying the remaining out-edges to point at the supervertex of the edge's destination vertex. Finally, in phase four, all vertices send their adjacency lists to their supervertex, which merges them according to minimum weight. Vertices designated as supervertices return to phase one as regular vertices, while all other vertices vote to halt. The algorithm terminates when only unconnected vertices remain.

The implementation of DMST is over 1300 lines of code and uses custom vertex, edge, and message data types. We add only 4 lines of code to make DMST compatible with GiraphUC and an additional change of 4 more lines of code for Giraph async. As described in Chapter 3.3.5, these changes are simple and do not affect the algorithm's logic.

3.4.2.4 PageRank

PageRank is an algorithm that ranks webpages based on the idea that more important pages receive more links from other pages. Like in [72], we use the accumulative update PageRank [85] (Algorithm 3). All vertices start with a value of 0.0. At each superstep, a vertex u sets *delta* to be the sum of all values received from its in-edges (or 0.15 in the first superstep), and its PageRank value $pr(u)$ to be $pr(u) + delta$. It then sends $0.85 \cdot delta/\deg^{+}(u)$ along its out-edges, where $\deg^{+}(u)$ is u's outdegree. The algorithm terminates after a user-specified K supersteps, and each output $pr(u)$ gives the expectation value for a vertex u. The probability value can be obtained by dividing the expectation value by the number of vertices.

Algorithm 3 PageRank pseudocode.

```
procedure COMPUTE(vertex, incoming messages)
    if superstep == 0 then
        vertex.setValue(0)
        $delta \leftarrow 0.15$
    else
        $delta \leftarrow$ sum of all message values
    vertex.setValue(vertex.currValue() + $delta$)
    if superstep $\leq K$ then
        $m \leftarrow$ number of outgoing edges of vertex
        Send $0.85 \cdot delta/m$ to all outgoing neighbours
    else
        voteToHalt()
```

All systems except GraphLab async terminate after a fixed number of (logical) supersteps, as this provides the best accuracy and performance. GraphLab async, which has no notion of supersteps, terminates after the PageRank value of every vertex u changes by less than a user-specified threshold ϵ between two consecutive executions of u.

3.4.3 Results

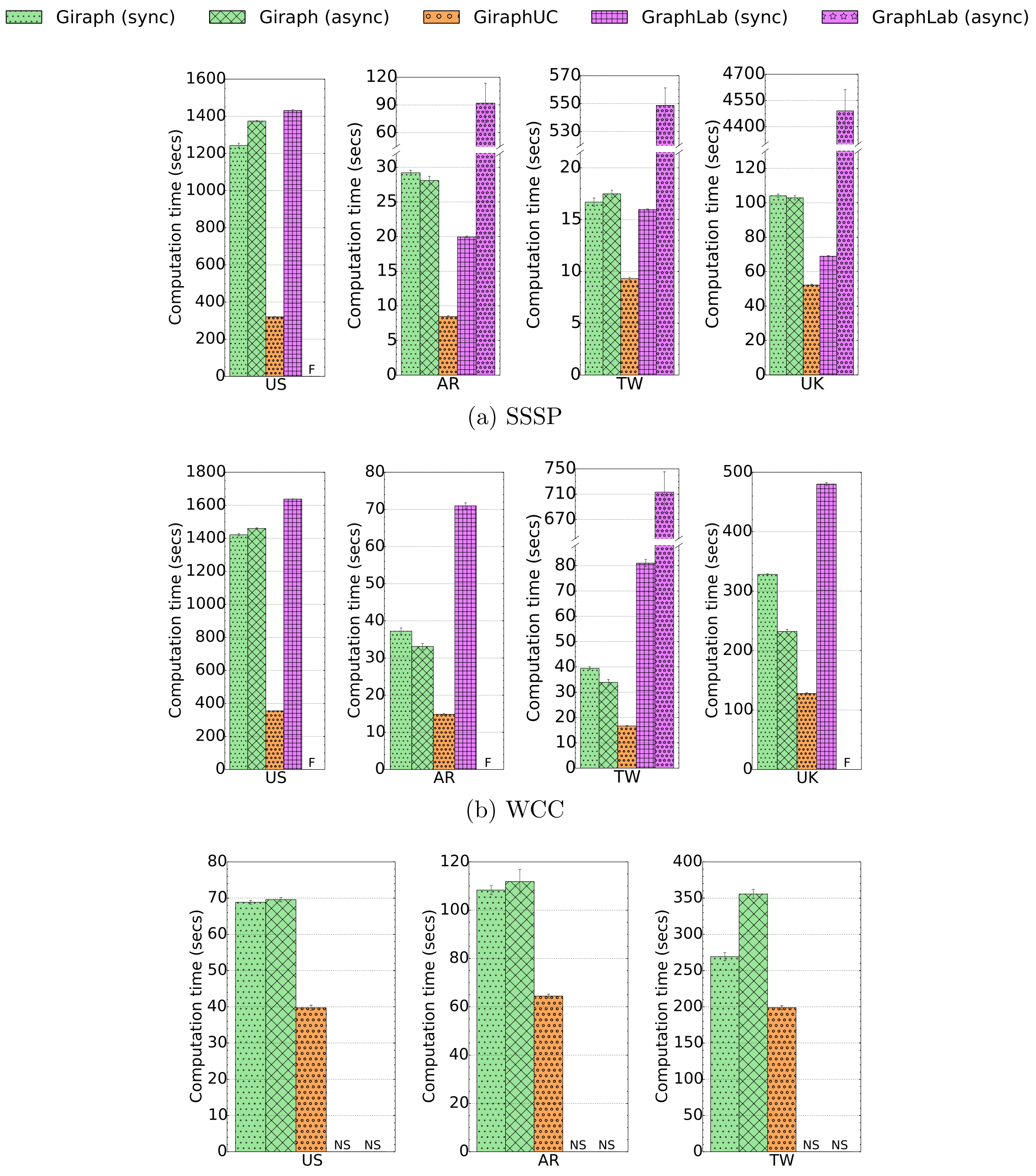
For our results, we focus on computation time, which is the total time of running an algorithm minus the input loading and output writing times. Computation time hence includes time spent on vertex computation, barrier synchronization, and network communication. This means, for example, it captures network performance: poor utilization of network resources translates to poor (longer) computation time. Since computation time captures everything that is affected by using different computation models, it accurately reflects the performance differences between each system.

For SSSP, WCC, and DMST (Figure 3.6), we report the mean and 95% confidence intervals of five runs. For PageRank (Figure 3.7), each data point is the mean of five runs, with 95% confidence intervals shown as vertical and horizontal error bars for both accuracy and time. Additionally, we ensure correctness by comparing the outputs of GiraphUC and Giraph async to that of Giraph sync. In total, we perform over 700 experimental runs.

3.4.3.1 SSSP

GiraphUC outperforms all of the other systems for SSSP on all datasets (Figure 3.6a). This performance gap is particularly large on `US`, which requires thousands of supersteps to complete due to the graph's large diameter (Table 3.1). By reducing per-superstep overheads, GiraphUC is up to 4.5× faster than Giraph sync, Giraph async, and GraphLab sync. Giraph async performs poorly due to the high per-superstep overheads of using global barriers. GraphLab async fails on `US` after 2 hours (7200s), indicated by an 'F' in Figure 3.6a, due to machines running out of memory during its distributed consensus termination. This demonstrates that GiraphUC's two step termination check has superior scalability.

On `AR`, `TW`, and `UK`, GiraphUC continues to provide gains. For example, it is up to 3.5× faster than Giraph sync, Giraph async, and GraphLab sync on `AR`. GraphLab async successfully runs on these graphs but its computation times are highly variable (Figure 3.6a) due to highly variable network overheads. These overheads are due to GraphLab async's lack of message batching and its pairing of fibers to individual vertices (Chapter 2.1.3), which results in highly non-deterministic execution compared to GiraphUC's approach of pairing compute threads with partitions (Chapter 3.3.1). GraphLab async's poor scalability



(a) SSSP

(b) WCC

(c) DMST

Figure 3.6: Computation times for SSSP, WCC, and DMST. Missing bars are labelled with ‘F’ for unsuccessful runs and ‘NS’ for unsupported algorithms.

is especially evident on TW and UK, where GiraphUC outperforms it by 59× and 86× respectively. Hence, GiraphUC is more scalable and does not suffer the communication overheads caused by GraphLab async's lack of message batching and its use of distributed locking and distributed consensus termination.

3.4.3.2 WCC

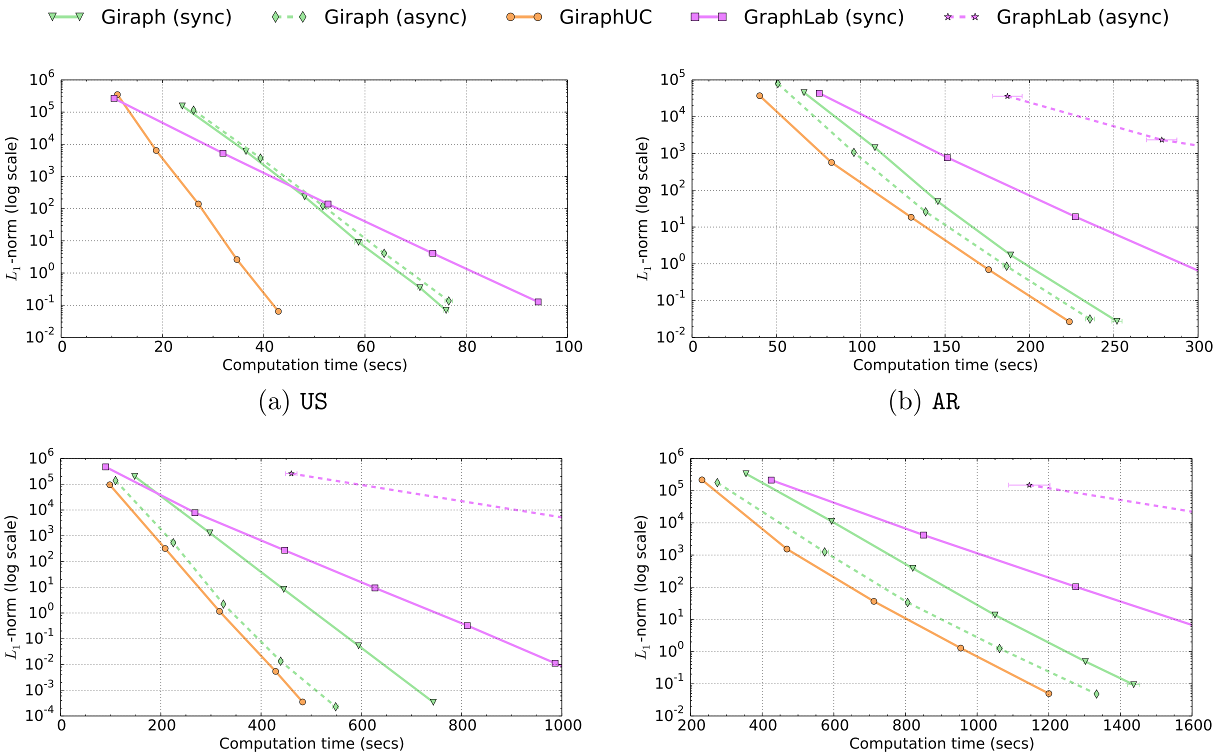
For WCC, GiraphUC consistently outperforms all of the other systems on all graphs: up to 4× versus Giraph sync and async on US, and nearly 5× versus GraphLab sync on TW (Figure 3.6b). In particular, whenever Giraph async has gains over Giraph sync, such as on UK, GiraphUC further improves on Giraph async's performance. In cases where Giraph async performs poorly, such as on US, GiraphUC still performs better than Giraph sync. This shows that the BAP model implemented by GiraphUC provides substantial improvements over the eAP model used by Giraph async.

Finally, like in SSSP, GraphLab async again performs poorly at scale: it fails on US after 5 hours (18,000s), AR after 20 minutes (1200s), and UK after 40 minutes (2400s) due to exhausting the available memory at several worker machines. For TW, on which GraphLab async successfully runs, GiraphUC is still 43× faster (Figure 3.6b).

3.4.3.3 DMST

For DMST, GiraphUC is 1.7× faster than both Giraph sync and async on US and AR, and 1.4× and 1.8× faster than Giraph sync and async respectively on TW (Figure 3.6c). These performance gains are primarily achieved in the second computation phase of DMST, which typically requires many supersteps to complete (Chapter 3.4.2.3). GiraphUC's gains are slightly lower than in SSSP and WCC because DMST is more compute bound, which means proportionally less time spent on communication and barriers. This is particularly true for TW, whose extreme degree skew leads to more computation time spent performing graph mutations. Nevertheless, GiraphUC's good performance establishes its effectiveness also for compute bound algorithms and algorithms that require multiple computation phases.

Giraph sync, Giraph async, and GiraphUC, when running DMST on UK, all exhaust the memory of several worker machines due to the size of the undirected weighted version of the graph. However, we expect trends to be similar since UK has a less extreme degree skew than TW (Table 3.1), meaning DMST will be less compute bound and can hence benefit more under GiraphUC.



(a) US

(b) AR

(c) TW

(d) UK

Figure 3.7: Plots of L_1-norm (error) vs. computation time for PageRank.

Note that GraphLab (both sync and async) cannot implement DMST as they do not fully support graph mutations. This is indicated in Figure 3.6c with 'NS' for "not supported". Hence, GiraphUC is both performant and more versatile with full support for graph mutations.

3.4.3.4 PageRank

PageRank, unlike the other algorithms, has a dimension of accuracy in addition to time. Like in [72], we define accuracy in terms of the L_1-norm between the output PageRank vector (the set of output vertex values) and the true PageRank vector, which we take to be the PageRank vector returned after 300 supersteps of synchronous execution [72]. The lower the L_1-norm, the lower the error and hence higher the accuracy. We plot the L_1-norm (in log scale) versus computation time to characterize performance in terms of both accuracy and time (Figure 3.7).

In the plots, all lines are downward sloping because the L_1-norm decreases (accuracy increases) with an increase in time, since executing with more supersteps or a lower ϵ tolerance requires longer computation times. In particular, this shows that Giraph async and GiraphUC's PageRank vectors are converging to Giraph's, since their L_1-norm is calculated with respect to Giraph's PageRank vector after 300 supersteps. When comparing the different lines, the line with the best performance is one that (1) is furthest to the left or lowest along the y-axis and (2) has the steepest slope. Specifically, (1) means that a fixed accuracy is achieved in less time or better accuracy is achieved in a fixed amount of time, while (2) indicates faster convergence (faster increase in accuracy per unit time).

From Figure 3.7, we see that GiraphUC has the best PageRank performance on all datasets. Its line is always to the left of the lines of all other systems, meaning it achieves the same accuracy in less time. For example, on `US` (Figure 3.7a), GiraphUC is 2.3× faster than GraphLab sync and 1.8× faster than Giraph sync in obtaining an L_1-norm of 10^{-1}. Compared to Giraph async, GiraphUC's line is steeper for `US` and `TW` and equally steep for `AR` and `UK`, indicating GiraphUC has similar or better convergence than Giraph async.

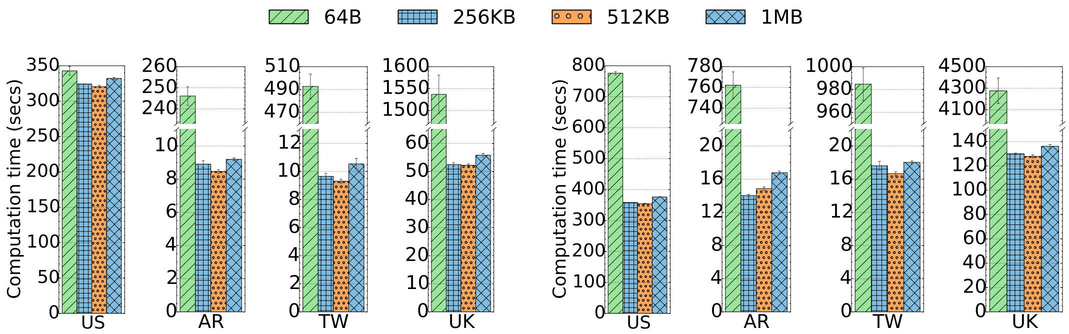
Lastly, GraphLab async again performs poorly due to limited scalability and communication overheads: its line is far to the right and has a very shallow slope (very slow convergence). Additionally, as observed with SSSP and WCC, its computation times tend to be highly variable: its horizontal (time) error bars are more visible than that of the other systems, which are largely obscured by the data point markers (Figures 3.7b and 3.7d). On `US`, GraphLab async achieves an L_1-norm of 2.6×10^5 after roughly 530s, which is 45× slower than GiraphUC. On `TW`, GraphLab async reaches an L_1-norm of 1.0 after roughly 3260s, meaning GiraphUC is 10× faster in obtaining the same level of accuracy.

3.4.4 Sensitivity Analysis

Lastly, we analyze the sensitivity of message batching and the performance of the naive vs. improved approach to local barriers in GiraphUC. All results are again the mean of five runs with 95% confidence intervals.

3.4.4.1 Message Batching

GiraphUC uses message batching to improve network utilization (Chapter 3.3.2). The amount of batching is controlled by the message buffer cache size, which is 512KB by default. Figure 3.8 shows how varying the buffer cache size from 64 bytes to 256KB, 512KB, and 1MB affects computation time.



(a) SSSP

(b) WCC

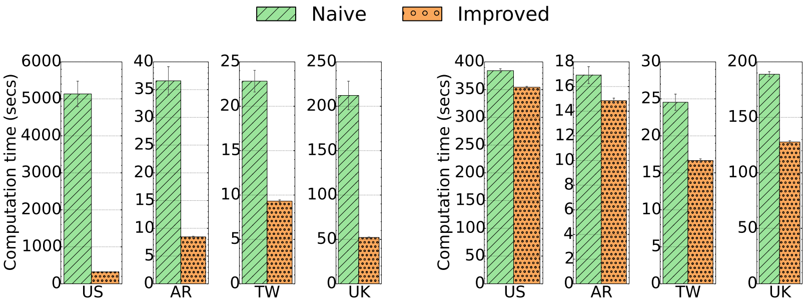
Figure 3.8: Computation times for SSSP and WCC in GiraphUC with different buffer cache sizes.

The buffer size of 64 bytes simulates a lack of message batching. This incurs substantial network overheads and long computation times: up to 53× slower than 512KB for SSSP on TW (Figure 3.8a). For SSSP on US, the performance deterioration is not as pronounced due to SSSP starting with a single active vertex combined with the large diameter of US: workers run out of work fairly quickly irrespective of the buffer cache size, meaning that the bulk of the network overheads are also incurred when using larger buffer sizes.

The buffer cache sizes of 256KB and 1MB demonstrate that the default 512KB is an optimal buffer size in that performance does not significantly improve with deviations from the default buffer size. This also indicates that dynamic tuning will likely provide minimal performance benefits. For WCC on AR, the 256KB buffer size performs slightly better than 512KB (Figure 3.8b). Upon further examination, the performance at 128KB and 64KB is identical to 256KB, but performance at 32KB is worse than at 512KB. Hence, even in this case the optimal range is large (between 64KB to 256KB) and using 512KB does not dramatically impact performance. Hence, we stay with the default 512KB buffer cache size for GiraphUC.

3.4.4.2 Local Barriers

As described in Chapter 3.2.1.2, the improved approach to local barriers is essential in making BAP efficient and GiraphUC performant. Compared to the naive approach, the improved approach is up to 16× faster for SSSP on US and 1.5× faster for WCC on TW and UK (Figure 3.9). Furthermore, the naive approach leads to higher variability in computation times because whether or not a worker blocks on a global barrier depends heavily on the timing of message arrivals, which can vary from run to run. In contrast, by allowing



(a) SSSP (b) WCC

Figure 3.9: Computation times for SSSP and WCC using naive vs. improved approach to local barriers.

workers to unblock, the improved approach suppresses this source of unpredictability and enables superior performance in GiraphUC.

3.5 Summary

In this chapter, we presented a new barrierless asynchronous parallel (BAP) computation model, which improves upon the existing BSP and AP models by reducing both message staleness and the frequency of global synchronization barriers. We showed how the BAP model supports algorithms that require graph mutations as well as algorithms with multiple computation phases, and also how the AP model can be enhanced to provide similar algorithmic support. We demonstrated how local barriers and logical supersteps ensure that each computation phase is completed using only one global barrier, which significantly reduces per-superstep overheads.

We described GiraphUC, our implementation of the BAP model in Giraph, a popular open source distributed graph processing system. Our extensive experimental evaluation showed that GiraphUC is much more scalable than GraphLab async and that it is up to 5× faster than Giraph, Giraph async, and GraphLab sync, and up to 86× faster than GraphLab async. Thus, GiraphUC enables developers to program their algorithms for the BSP model and transparently execute using the BAP model to maximize performance.

Chapter 4

Providing Serializability

We begin in Chapter 4.1 by motivating serializability using a concrete example of graph coloring. In Chapter 4.2 and 4.3, we formalize serializability and describe both existing techniques and our partition-based approach. In Chapter 4.4, we detail the implementations of these techniques in Giraph async and GiraphUC. We present an extensive experimental evaluation in Chapter 4.5 before concluding with a summary in Chapter 4.6.

4.1 Motivation

4.1.1 BSP Model

For our concrete example, consider the greedy graph coloring algorithm (Chapter 4.5.2.1). Each vertex starts with the same color (denoted by its vertex value) and, in each superstep, selects the smallest non-conflicting color based on its received messages, broadcasts this change to its neighbours, and votes to halt. The algorithm terminates when there are no more color conflicts.

Consider an undirected graph of four vertices partitioned across two worker machines (Figure 4.1). All vertices broadcast the initial color 0 in superstep 1 but the messages are not visible until superstep 2. Consequently, in superstep 2, all vertices update their colors to 1 based on stale data. Similarly for superstep 3. Hence, vertices collectively oscillate between 0 and 1 and the algorithm never terminates. However, if we could ensure that only v_0 and v_3 execute in superstep 2 and only v_2 and v_1 execute in superstep 3, then this problem would be avoided. As Chapter 4.3.3.1 will show, serializability provides precisely this solution.

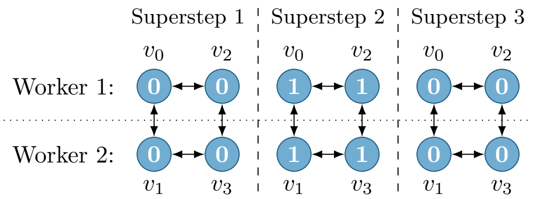


Figure 4.1: BSP execution of greedy graph coloring. Each graph is the state at the end of that superstep.

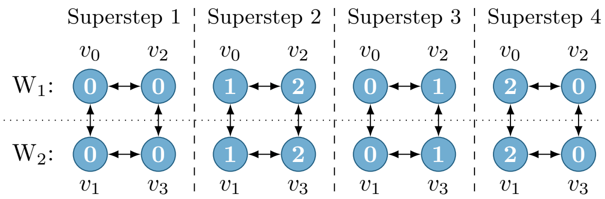

Superstep 1
Superstep 2
Superstep 3
Superstep 4
v_0 v_2 v_0 v_2 v_0 v_2 v_0 v_2
W_1:
0 0 1 2 0 1 2 0
W_2:
0 0 1 2 0 1 2 0
v_1 v_3 v_1 v_3 v_1 v_3 v_1 v_3

Figure 4.2: AP execution of greedy graph coloring. Each graph is the state at the end of that superstep.

4.1.2 AP Model

Like BSP, the AP model[1] can also fail to terminate for the greedy graph coloring algorithm. Consider again the undirected graph (Figure 4.2) and suppose that workers W_1 and W_2 execute their vertices sequentially as v_0 then v_2 and v_1 then v_3, respectively. Furthermore, suppose the pairs v_0, v_1 and v_2, v_3 are each executed in parallel. Then the algorithm fails to terminate. Specifically, in superstep 2, v_0 and v_1 update their colors to 1 and broadcast 1 (we reserve superstep 1 for initialization). v_0 and v_1 do not see each other's message until superstep 3. v_2 and v_3, however, see this 1 along with the message 0 that they sent to each other in superstep 1. Thus, v_2 and v_3 update their colors to 2 and broadcast. Ultimately, the graph's state at superstep 5 is the same as in superstep 2, so vertices are collectively cycling through the three colors in an infinite loop.

However, if we can force v_0 to execute concurrently with v_3 instead of v_1 (and v_2 with v_1) while also ensuring that v_2 (v_1) sees v_3's (v_0's) message in the same superstep that it is sent, then the algorithm will terminate. In fact, it does so in two supersteps, fewer than the three required by BSP. We formalize this intuition in Chapters 4.2 and 4.3.

4.1.3 BAP Model

As described in Chapter 3.2, the BAP model improves on the AP model by minimizing the number of global barriers via per-worker logical supersteps separated by local barriers. However, like AP, BAP can also fail to terminate when executing the greedy graph coloring algorithm.

We include BAP because it provides significant performance gains for algorithms such as SSSP that require many supersteps (Chapter 3.4.3). Furthermore, BAP is a transparent system-level execution model, so algorithms written for BSP (and AP) work with little to

[1] Technically, we must use the eAP model and have new messages overwrite old messages (Theorem 2).

no modifications (Chapter 3.2). In fact, the techniques for providing serializability are identical for both the AP and BAP models (Chapter 4.3).

4.1.4 GAS Model

As described in Chapter 2.1.3, sync GAS is similar to BSP and so fails to terminate in the same way (Figure 4.1), while async GAS can fail to terminate in the case of dense graphs [34]. For example, for the graph coloring example in Figure 4.2, suppose both W_1 and W_2 each have two threads for their two vertices and all threads run in parallel. Then the interleaving of the GAS phases will cause vertices to see stale colors and so the execution is not guaranteed to terminate: it can become stuck in an infinite loop. In contrast, by adding the vertex-based distributed locking technique (Chapter 2.1.3), async GAS with serializability will always terminate successfully.

4.2 Serializability

In this section, we formally define serializability and prove several key properties.

4.2.1 Preliminaries

Existing work [34, 71] considers serializability for algorithms where vertices communicate only with their direct neighbours, which is the behaviour of the majority of algorithms that require serializability. For example, the GAS model supports only algorithms where vertices communicate with their direct neighbours [50, 34]. For these algorithms, serializability can be provided transparently by the graph processing system, independent of the algorithm being executed. Thus, our focus is on these types of algorithms.

Since popular graph processing systems use a vertex-centric programming model, where developers specify the actions of a single vertex, we focus on vertex-centric systems. The formalisms that we will establish apply to all vertex-centric systems, irrespective of the computation models they use.

In vertex-centric graph processing systems, there are two levels of parallelism: (1) between multiple threads within a single worker machine and (2) between the multiple worker machines. Due to the distributed nature of computation, the input graph must be partitioned across the workers and so data replication will occur. To better understand this, let *neighbours* refer to both in-edge and out-edge neighbours.

Definition 1. *A vertex u is a* machine boundary *vertex, or* m-boundary *for short, if at least one of its neighbours v belongs to a different worker machine from u. Otherwise, u is a* machine internal*, or* m-internal*, vertex.*

Definition 2. *A replica is* local *if it belongs to the same worker machine as its primary copy and* remote *otherwise.*

Systems keep a read-only replica of each vertex on its owner's machine and of each m-boundary vertex u on each of u's out-edge neighbour's worker machines. This is a standard design used, for example, in Pregel, Giraph, and GraphLab. Remote replicas (of m-boundary vertices) exist due to graph partitioning: for vertex-cut partitioning, u's vertex value is explicitly replicated on every out-edge neighbour v's worker machine; for edge-cut, u is implicitly replicated because the message it sends to v, which is a function of u's vertex value, is buffered in the message store of v's machine. This distinction is unimportant for our formalism as we care only about whether replication occurs. Local replicas occur in push-based systems because message stores also buffer messages sent between vertices belonging to the same worker. In pull-based systems, local replicas are required for implementing synchronous computation models like sync GAS. For asynchronous models, pull-based systems may not always have local replicas (such as in GraphLab async) but we will consider the more general case in which they do (if they do not, then reads of such vertices will always trivially see up-to-date data).

Definition 3. *A read of a replica is* fresh *if the replica is up-to-date with its primary copy and* stale *otherwise.*

Informally, an execution is serializable if it produces the same result as a serial execution in which all reads are fresh. More formally, we require one-copy serializability (1SR) [13].

Definition 4. *A system provides* serializability*, or* serializable executions*, if and only if all executions produce histories that are one-copy serializable (1SR).*

In the subsequent sections, we prove that 1SR can be provided by enforcing two conditions: (1) all vertices read up-to-date replicas of its in-edge neighbours prior to execution and (2) no vertex has executions that are concurrent with any of its in-edge neighbours. In terms of traditional transaction terminology, we define a site as a worker machine, an item as a vertex, and a transaction as the execution of a single vertex. We detail such transactions next.

4.2.2 Transactions

We define a transaction to be the single execution of an arbitrary vertex u, consisting of a read on u and the replicas of u's in-edge neighbours followed by a write to u. The read acts only on u and its in-edge neighbours because u receives messages (or pulls data) from only its in-edge neighbours—it has no dependency on its out-edge neighbours. Denoting the read set as $N_u = \{u, u\text{'s in-edge neighbours}\}$, any execution of u is the transaction $T_i = r_i[N_u]w_i[u]$, or simply $T_i(N_u)$ as all transactions are of the same form.

Any $v \in N_u$ with $v \neq u$ is also annotated to distinguish it from the other read-only replicas of v. For example, if u belongs to worker A, we annotate the read-only replica as $v_A \in N_u$. However, the next section will show how we can drop these annotations.

Our definition relies only on the fact that the system is vertex-centric and not on the nuances of specific computation models. For example, although BSP and AP have a notion of supersteps, the i for a transaction $T_i(N_u)$ has no relation to the superstep count. The execution of u in two different supersteps is represented by two different transactions $T_i(N_u)$ and $T_j(N_u)$. Our definitions also work when there is no notion of supersteps, such as in the async GAS model, or when there are only per-worker logical supersteps, such as in the BAP model.

We can now restate our two conditions more formally as:

Condition C1. *Before any transaction $T_i(N_u)$ executes, all replicas $v \in N_u$ on u's machine are up-to-date.*

Condition C2. *No transaction $T_i(N_u)$ overlaps with any transaction $T_j(N_v)$ for all copies of $v \in N_u$, $v \neq u$.*

4.2.3 Replicated Data

We first show how to enforce condition C1 and then prove, in Lemma 1, that enforcing condition C1 simplifies the problem to standard serializability on a single logical copy of each vertex (i.e., without data replication).

We enforce condition C1 by combining a *write-all* approach [13] with a *synchronization technique* that ensures neighbouring vertices do not execute concurrently. Denote u's in-edge and out-edge neighbours as $\mathcal{M}_u$. Then a synchronization technique prevents stale reads by ensuring no transactions $T_j(N_v)$ with any copy of $v \in \mathcal{M}_u$ occur while updates are being propagated to u's replicas. In theory, the write-all approach must update replicas eagerly: $T_i(N_u)$ commits only after all replicas of u are synchronously updated. In practice, vertices coordinated by a synchronization technique can update their replicas lazily, because any such vertex v must first acquire a shared resource (e.g., a token or a fork) from its neighbour u before it can execute. Thus, updates to u's replicas can be delayed and instead sent before handing over the resource required by v. The effect is the same as an eager write-all approach: v always reads an up-to-date replica of u. Chapter 4.3.1.1 describes this approach in greater detail.

The write-all approach works for graph processing systems because we replicate for distributed computation, not for availability. That is, when a worker machine fails, we lose a portion of our input graph and so we cannot proceed with the computation. Indeed, failure recovery requires all machines to rollback to a previous checkpoint [1, 50, 54], meaning pending writes to failed machines do not occur.

Lemma 1. *If condition C1 is true, then it suffices to use standard serializability theory where operations are performed on a single logical copy of each vertex.*

Proof. Condition C1 ensures that before every transaction $T_i(N_u)$ executes, the replicas $v \in N_u$ are all up-to-date. Then all reads $r_i[N_u]$ see up-to-date replicas and are thus the same as reading from the primary copy of each $v \in N_u$. Hence, there is effectively only a single logical copy of each vertex, so we can apply standard serializability theory. □

4.2.4 Correctness

We use the standard notion of conflicts and histories [13]. Two transactions conflict if they have conflicting operations. We assume, per Lemma 1, that reads and writes are performed on a single logical copy of each vertex.

A *serial single-copy history* produced by the serial execution of an algorithm is a sequence of transactions where operations act on a single logical copy and do not interleave. For example, one possible serial single-copy history for two vertices u and v would be

$$r_1[N_u]w_1[u]c_1r_2[N_v]w_2[v]c_2r_3[N_u]w_3[u]c_3r_4[N_v]w_4[v]c_4,$$

where c_i denotes the commit of each transaction. Irrespective of the computation model, the history indicates that u and v are each executed twice before the algorithm terminates. If the computation model has a notion of supersteps, and we assume that u and v are executed exactly once in each superstep, then T_1 and T_2 would belong to the first superstep while T_3 and T_4 would belong to the second.

As per Definition 4, we require 1SR. We first prove a useful relationship between input and serializability graphs (Lemma 2) before formally defining the relationship between serializability and conditions C1 and C2 (Theorem 4).

Lemma 2. *Suppose condition C1 is true. Then a directed cycle of ≥ 2 vertices exists in the input graph if and only if there exists an execution that produces a serialization graph containing a directed cycle.*

Proof. Since condition C1 holds, by Lemma 1 we can apply standard serializability theory.

(Only if) Suppose the input graph contains a directed cycle of $n \geq 2$ vertices and, without loss of generality, assume that the cycle is formed by the edges (u_n, u_1) and (u_i, u_{i+1}) for $i \in [1, n)$. Consider executions in which all read operations occur before any write operation. That is, consider histories of the form $r_1[N_{u_1}] \cdots r_n[N_{u_n}] w_1[u_1] c_1 \cdots w_n[u_n] c_n$ where operations can be arbitrarily permuted so long as all reads precede all writes. This does not violate condition C1 because all reads are fresh.

Recall that the serialization graph of a history consists of a node for each transaction and an edge from T_i to T_j if one of T_i's operations precedes and conflicts with one of T_j's operations [13]. Since all reads occur before writes, every transaction T_i's read operation $r_i[N_{u_i}]$ will precede and conflict with transaction T_{i-1}'s write operation $w_{i-1}[u_{i-1}]$ for $i \in (1, n]$ because $u_{i-1} \in N_{u_i}$. That is, there exists a directed path from T_n to T_1 in the serialization graph. Additionally, because of the directed cycle in the input graph, we also have a conflict due to $u_n \in N_{u_1}$. This adds an edge from T_1 to T_n which, together with the directed path from T_n to T_1, creates a directed cycle in the serialization graph. Hence, all histories of this form produce serialization graphs with a directed cycle.[2]

(If) For the converse, suppose there exists a cycle in the serialization graph. Then there exist two transactions $T_i(N_u)$ and $T_j(N_v)$, for arbitrary i, j and vertices u, v, that must be ordered as $T_j(N_v) < T_i(N_u)$ and $T_i(N_u) < \cdots < T_j(N_v)$.

The former ordering can only be due to $r_j[N_v] < w_i[u]$, since every transaction has its read precede its write, which implies that $u \in N_v$ and so there exists an edge (u, v) in the input graph. Similarly, the latter ordering implies that there is a directed path from v to u. For example, suppose the ordering is $T_i(N_u) < T_k(N_w) < T_j(N_v)$. Then this must arise

[2] Another proof can also be achieved using induction. For example, histories of the form

$$r_n[N_{u_n}] r_{n-1}[N_{u_{n-1}}] \cdots r_1[N_{u_1}] w_1[u_1] c_1 \cdots w_n[u_n] c_n$$

will always require $T_n < T_{n-1} < \cdots < T_1$, because $u_i \in N_{u_{i+1}}$ for $i \in [1, n)$, and $T_1 < T_n$ since $u_n \in N_{u_1}$. The two orderings then create a cycle in the serialization graph.

due to $r_i[N_u] < w_k[w]$ and $r_k[N_w] < w_j[v]$, meaning $w \in N_u$ and $v \in N_w$. Equivalently, there exist edges (w, u) and (v, w) in the input graph and thus a directed path from v to u. In general, the more transactions there are between $T_i(N_u)$ and $T_j(N_v)$, the longer this path.

Since there is both a directed path from v to u and an edge (u, v), the input graph has a directed cycle. □

Theorem 4. *All executions are serializable for all input graphs if and only if conditions C1 and C2 are both true.*

Proof. (If) Since condition C1 is true, by Lemma 1 we can apply standard serializability theory. By the serializability theorem [13], it suffices to show if condition C2 is true then all possible histories have an acyclic serialization graph.

Consider two arbitrary concurrent transactions $T_i(N_u)$ and $T_j(N_v)$. For vertex-centric systems, each vertex is always executed by exactly one thread of execution because the compute function is written for serial execution. That is, we must have $u \neq v$ for T_i and T_j. Since condition C2 is true, the write operations of T_i and T_j are always on $u \notin N_v$ and $v \notin N_u$, respectively, meaning the transactions do not conflict. That is, condition C2 eliminates all conflicting transactions from all possible histories, so the serialization graph must be acyclic. Since we make no assumptions about the input graph, this holds for all input graphs.

(Only if) Next, we prove the inverse: if either condition C1 or C2 is false, then there exists a non-serializable execution for some input graph. Consider an input graph with vertices u and v connected by an undirected edge and executed by the transactions $T_1(N_u)$ and $T_2(N_v)$, respectively.

Suppose C1 is true but C2 is not. Then $T_1(N_u)$ and $T_2(N_v)$ can execute in parallel and so a possible history is $r_1[N_u]r_2[N_v]w_1[u]c_1w_2[v]c_2$. This history does not violate C1 because both reads see up-to-date state. However, the two transactions conflict: $v \in N_u$ implies $T_1 < T_2$ while $u \in N_v$ implies $T_2 < T_1$, so this execution is not serializable. More generally, per Lemma 2, this occurs for any graph with a directed cycle of ≥ 2 vertices.

Suppose C2 is true but C1 is not. Then standard serializability no longer applies as there are now replicas. For our input graph, let u be at worker A and v be at worker B. Then $N_u = \{u, v_A\}$ and $N_v = \{v, u_B\}$, where v_A and u_B are replicas of v and u respectively. Consider a serial history $r_1[\{u, v_A\}]w_1[u]c_1r_2[\{v, u_B\}]w_2[v]c_2$ where initially $v_A = v$ and $u_B = u$, but $u_B \neq u$ after T_1. This history is possible under condition C2 (it is serial) but it is not 1SR: $v_A = v$ implies $T_1 < T_2$ while $u_B \neq u$ implies $T_2 < T_1$, so there are no conflict equivalent serial single-copy histories. Another way to see this is that the above history is effectively a concurrent execution of T_1 and T_2 on a single copy of u and v—i.e., where condition C1 is true but C2 is not. □

Lastly, we note a theoretically interesting corollary.

Corollary 1. *Suppose condition C1 is true. Then all executions are serializable if and only if the input graph has no directed cycles of* ≥ 2 *vertices.*

Proof. This is just the contrapositive of Lemma 2: there are no directed cycles of ≥ 2 vertices in the input graph iff there are no executions that produce a serialization graph containing a cycle. Applying the serializability theorem, we have: all executions are serializable iff the input graph has no directed cycles of ≥ 2 vertices.[3] □

However, in practice, Corollary 1 is insufficient for providing serializability because nearly all real world graphs have at least one undirected edge. Moreover, the synchronization technique used to enforce condition C1 will also enforce condition C2 (Chapters 4.2.3 and 4.2.5).

4.2.5 Enforcing Serializability

Graph processing systems that implement any of the computation models from Chapter 2.1 do not enforce conditions C1 and C2 as none of the models utilize any synchronization techniques. Hence, by Theorem 4, they do not provide serializability. Moreover, these systems do not guarantee fresh reads even under serial executions (on a single machine or under the sequential execution of multiple machines). For example, BSP effectively updates replicas lazily[4] because messages sent in one superstep, even if received, cannot be read by the recipient in the same superstep. Thus, both m-boundary *and* m-internal vertices (Definition 1) suffer stale reads under a serial execution. While AP weakens this isolation and can update local replicas eagerly, it propagates messages to remote replicas lazily without the guarantees of condition C1 and so stale reads can again occur under a serial execution of multiple machines.

[3] Self-cycles in the input graph are permitted because a single vertex is executed by one thread of execution at any time (see the proof of Theorem 4).

[4] The “synchronous” in BSP refers to the global communication barriers, *not* the method of replica synchronization.

To provide serializability, we enforce conditions C1 and C2 by adding a synchronization technique (Chapter 4.3) to the systems that implement the above computation models. For example, GraphLab async provides serializability by adding a distributed locking synchronization technique on top of the async GAS model. These synchronization techniques also implement a write-all approach for updating replicas, which is required for condition C1.

The synchronization techniques ensure that a vertex u does not execute concurrently with any of its in-edge *and* out-edge neighbours. At first glance, this appears to be stronger than what condition C2 requires. However, suppose v is an out-edge neighbour of u and v is currently executing. Then if u does not synchronize with its out-edge neighbours, it will erroneously execute concurrently with v, violating condition C2 for v. Alternatively, if v is an out-edge neighbour of u then u is an in-edge neighbour of v, so they must not execute concurrently.

4.3 Synchronization Techniques

In this section, we describe synchronization techniques to enforce conditions C1 and C2 as well as their performance with respect to parallelism and communication overheads.

4.3.1 Preliminaries

4.3.1.1 Replica Updates

How a synchronization technique implements the write-all approach (Chapter 4.2.3) depends on whether the computation model is synchronous or asynchronous.

For asynchronous computation models (AP, BAP, async GAS), replicas immediately apply received updates. Thus, local replicas can be updated eagerly, since there are no communication costs (Chapter 4.4). Remote replicas, however, are updated lazily in a just-in-time fashion to provide communication batching: if an m-boundary vertex u has a replica on its neighbour v's worker then, when v wants to execute, u's worker will flush all pending remote replica updates before handing over the shared resource that allows v to proceed.

In contrast, synchronous computation models (BSP, sync GAS) hide updates from replicas until the next superstep. That is, replicas can only be updated after a global barrier. This means systems with synchronous models are limited to specialized synchronization techniques that keep replicas up-to-date by dividing each superstep into multiple sub-supersteps (Proposition 2). This is significantly less performant than synchronization techniques for systems with asynchronous models (Chapter 4.4).

4.3.1.2 Architectural Considerations

In addition to the type of computation model used, the synchronization techniques that a graph processing system can support is also limited by the system's architectural design and, in particular, whether it is partition-aware.

GraphLab async over-threads so that it can pair lightweight threads (called fibers) with individual vertices. This ensures CPU cores are kept busy even when some fibers need to block on network communication and is well-suited for fine-grained synchronization techniques. However, GraphLab async is not partition-aware. In contrast, Giraph assigns multiple graph *partitions* to each worker and pairs compute threads, each roughly equivalent to a CPU core, with available partitions. Thus, the smallest unit of serial execution is a partition of many vertices, meaning Giraph is better suited for coarse-grained techniques and is also partition-aware. This also applies to Giraph async and GiraphUC, as they are built on top of Giraph (Chapter 3.3).

In the following sections, we show how the differences between the two designs affect performance and why it is useful for systems to be partition-aware. As Giraph, Giraph async, and GiraphUC all share the same architectural design, we will refer to them collectively as Giraph's architecture.

4.3.2 Token Passing

The simplest synchronization technique is to pass an exclusive token around a logical token ring. We focus on two methods of token passing: the single-layer approach, considered in [71], and a dual-layer approach. We consider the more general case of Giraph's partition-aware architecture, where each worker executes its partitions in parallel with multiple threads. GraphLab async's architecture is the special case where each vertex is effectively in its own partition.

4.3.2.1 Single-Layer Token Passing

For single-layer token passing, vertices are categorized as either p-internal or p-boundary per the following definition.

Definition 5. *A vertex u is a* partition boundary *vertex, or* p-boundary *for short, if at least one of its neighbours v belongs to a different partition from u. Otherwise, u is a* partition internal*, or* p-internal*, vertex.*

A global token is passed in a round-robin fashion between partitions arranged in a logical ring. Each worker holds the global token for multiple iterations on behalf of each of its partitions to ensure that every partition's p-boundary vertices are executed.

It is easy to see that this prevents neighbouring vertices from executing concurrently: a p-internal vertex and its neighbours are all executed by a single thread, so there is no concurrency, while a p-boundary vertex can execute only when its partition holds an exclusive token. As per Chapter 4.3.1.1, local replicas are updated eagerly while remote replicas of a worker's m-boundary vertices are updated in batch before the worker passes along the global token. Remote replicas need not be updated when passing the token between partitions of the same worker because no other worker can execute vertices that read these replicas. Thus, single-layer token passing enforces conditions C1 and C2 for asynchronous computation models.

However, the single-layer approach is not suitable for multithreading because it is too coarse-grained: a vertex with neighbours in different partitions of the same worker can execute only when its partition holds the global token, even though it does not require coordination with vertices of other workers.

4.3.2.2 Dual-Layer Token Passing

To address the shortcomings of the single-layer approach, we introduce a dual-layer approach that uses two layers of tokens and a more fine-grained categorization of vertices.

Let u be a vertex of partition P_u of worker W_u. As before, u is a p-internal vertex if all its neighbours belong to P_u. A p-boundary vertex is now one of three types. Informally, u is a *local boundary* vertex if its neighbours are on partitions of W_u, *remote boundary* if its neighbours are on partitions of other workers, and *mixed boundary* if its neighbours belong to partitions of both W_u and other workers. More formally, let W_u and P_u be sets where $u \in P_u$ and $P_u \subseteq W_u \subseteq V$ (V being the set of all vertices). Denote u's in-edge and out-edge neighbours as $\mathcal{M}_u$. Then u is local boundary if $\mathcal{M}_u \subseteq W_u$ and $\exists v \in \mathcal{M}_u$ s.t. $v \notin P_u$, remote boundary if $\forall v \in \mathcal{M}_u$ either $v \in P_u$ or $v \notin W_u$, and mixed boundary otherwise (i.e., $\exists v, w \in \mathcal{M}_u$ s.t. $v \in W_u \setminus P_u$ and $w \notin W_u$).

A global token is passed in a round-robin fashion between workers rather than partitions. Each worker also has its own local token that is passed between its partitions in a round-robin fashion. A p-internal vertex can execute without tokens, while a local boundary vertex requires its partition to hold the local token and a global boundary vertex requires its worker to hold the global token. A mixed boundary vertex requires both tokens to be held. Like the single-layer approach, local replicas are updated eagerly while remote replicas are updated before a worker relinquishes the global token. Hence, dual-layer token passing also enforces conditions C1 and C2 for asynchronous computation models.

The dual-layer approach provides improved parallelism over the single-layer approach: local boundary vertices need not wait for the global token to execute, while remote boundary vertices can execute so long as its worker (rather than its partition) holds the global token. Like the single-layer approach, each worker holds the global token for a number of iterations equal to the number of partitions it owns to ensure that all mixed boundary vertices get to execute.

4.3.2.3 Discussion

Compared to Giraph's architecture, GraphLab async's architecture is a poor fit for token passing: pairing fibers with vertices means each vertex is effectively a partition, so all vertices are p-boundary. Thus, there is no parallelism within workers: each vertex requires token(s) to execute.

Token passing minimizes communication overheads at the cost of parallelism (Figure 1.2). It does not scale well because the size of the token ring increases with the number of partitions or workers, which leads to longer wait times. For the two approaches above, the rings are also fixed: workers or partitions that are finished must still receive and pass along the tokens. To complicate matters, in algorithms like SSSP, workers and partitions dynamically halt or become active depending on the state of their constituent vertices. Even with the dual-layer approach, token passing remains too coarse-grained, which restricts parallelism.

While these issues can be addressed with more sophisticated schemes, such as tracking additional state to support a dynamic ring or using multiple global tokens to increase parallelism, it becomes much harder to guarantee correctness (no deadlocks or starvation) while also ensuring fairness. Rather than make token passing more fine-grained, we present a more coarse-grained distributed locking synchronization technique next.

4.3.3 Partition-based Distributed Locking

Partition-based distributed locking builds on the Chandy-Misra algorithm [22], which solves the hygienic dining philosophers problem, a generalization of the traditional dining philosophers problem. Each philosopher is either thinking, hungry, or eating and must acquire a shared fork from each of its neighbours to eat. Philosophers can communicate with their neighbours by exchanging forks and request tokens for forks. The "dining table" is essentially an undirected graph where each vertex is a philosopher and each edge is associated with a shared fork. A philosopher u thus requires $\deg(u)$ forks to eat. The Chandy-Misra algorithm ensures no neighbouring philosophers eat at the same time and guarantees fairness (no philosopher can hog its forks) and no deadlock or starvation [22].

Partition-based distributed locking treats each partition as a philosopher. Two partitions share a fork if an edge connects their constituent vertices. More formally, partitions P_i and P_j share a fork if there exists a pair of neighbouring vertices $u \in P_i$ and $v \in P_j$. Then condition C2 is enforced for p-boundary vertices since neighbouring partitions never execute concurrently. p-internal vertices do not need coordination as each partition is executed serially. As an optimization, we can avoid unnecessary fork acquisitions by skipping partitions for which all vertices are halted and have no more messages.

To enforce condition C1, per Chapter 4.3.1.1, local replicas are updated eagerly and, for remote replicas, each worker flushes its pending remote replica updates before any partition (with an m-boundary vertex) relinquishes a fork to a partition of another worker. Since both conditions are enforced, Proposition 1 follows immediately.

Proposition 1. *Partition-based distributed locking enforces conditions C1 and C2 for asynchronous computation models.*

By Theorem 4, partition-based distributed locking provides serializability for asynchronous computation models. However, it is incompatible with synchronous computation models, which can update replicas only after a global barrier (Chapter 4.3.1.1), because p-internal vertices are executed sequentially and so we must update local replicas eagerly to enforce condition C1 (ensure fresh reads).

Partition-based distributed locking needs at most $O(|P|^2)$ forks, where $|P|$ is the total number of partitions. By controlling the number of partitions, we can control the granularity of parallelism. On one extreme, $|P| = |V|$ gives vertex-based distributed locking (Chapter 4.3.3.1). On the other extreme, we can have exactly one partition per worker. This still provides better parallelism than token passing because any pair of non-neighbouring partitions can execute in parallel, with a negligible increase in communication. In general,

$|P|$ is set such that each worker can use multiple threads to execute multiple partitions in parallel.

4.3.3.1 Vertex-based Distributed Locking

For GraphLab async, which pairs fibers with vertices, vertex-based distributed locking is a special case of partition-based locking where each vertex is in its own partition. However, for Giraph's partition-aware architecture, vertex-based locking can be further optimized: since p-internal vertices are executed sequentially, only p-boundary vertices need to act as philosophers and be coordinated using the Chandy-Misra algorithm. To update remote replicas, workers flush remote replica updates before any m-boundary vertex relinquishes a fork to a vertex of another worker. Thus, vertex-based locking also enforces conditions C1 and C2.

By Theorem 4, this solution provides asynchronous computation models with serializability. However, like partition-based locking, this solution is incompatible with synchronous models (BSP and sync GAS). Proposition 2 shows that a constrained vertex-based locking solution can provide serializability for systems with synchronous models.

Proposition 2. *Vertex-based distributed locking enforces conditions C1 and C2 for synchronous computation models when the following two properties hold: (i) both p-internal and p-boundary vertices act as philosophers and (ii) fork and token exchanges occur only during global barriers.*

Proof. By property (i), all vertices act as philosophers, so no neighbouring vertices can execute concurrently. Thus, condition C2 is enforced. For condition C1, it remains to show that all replicas are kept up-to-date. Property (i) ensures that p-internal vertices are also coordinated because local replicas can be updated only after a global barrier. Property (ii) ensures that, in each superstep, only a non-neighbouring subset of vertices are executed. For example, if we have an input graph with the edge (u, v), in each superstep either u executes or v executes but not both: u (v) cannot obtain the fork from v (u) in the same superstep because fork and token exchanges must occur only at a global barrier. This is required because replicas can be updated only after a global barrier: if u and v ran in the same superstep, one of the two will perform a stale read. Hence, condition C1 is enforced by effectively dividing each superstep into multiple sub-supersteps. □

Vertex-based distributed locking requires, in the worst case, $O(|\mathcal{E}|)$ forks, where $|\mathcal{E}|$ is the number of edges in the graph ignoring directions (i.e., counting undirected edges once).

Thus, compared to token passing and partition-based locking, it achieves more parallelism at a substantially higher communication cost of $O(|\mathcal{E}|)$ messages per iteration (Figure 1.2). Furthermore, while token passing scales poorly due to its ring size, vertex-based distributed locking scales poorly due to its communication overheads.

4.3.3.2 Discussion

For vertex-based distributed locking, GraphLab async's architecture is a better fit than Giraph's architecture, as Giraph blocks an entire CPU whenever a vertex blocks on communication. However, Giraph's architecture supports partitions, and thus partition-based distributed locking, while GraphLab async does not. As we show in Chapter 4.5.3, the superior performance of the partition-based approach demonstrates that it is important for systems to be partition-aware.

Additionally, in the vertex-based approach, large batches of messages (remote replica updates) are difficult to form as messages must be flushed very frequently due to the large number of forks (Chapter 4.4.4). While GraphLab async tries to mitigate this through the use of fibers to mask communication latency, it is more performant to pair a partition-aware system with a partition-based approach that better supports communication batching. Specifically, partition-based distributed locking can batch messages for an entire partition of vertices, which substantially reduces communication overheads. Finally, fork and token exchange messages cannot be batched under either approach. However, this has a much smaller impact on communication overheads for partition-based locking than vertex-based locking as the former uses far fewer forks. These factors further contribute to the much better performance of partition-based locking, even when vertex-based locking is paired with a tailored architecture such as GraphLab async (Chapter 4.5.3).

Hence, partition-based distributed locking leverages the best of both worlds: the increased parallelism of vertex-based distributed locking and the minimal communication overheads of token passing. It also scales better than vertex-based locking and token passing due to its lower communication overheads and the absence of a token ring. Finally, this solution offers flexibility in the number of partitions, allowing for a tunable trade-off between parallelism and communication overheads.

4.4 Implementation

We now describe our implementations for dual-layer token passing, partition-based distributed locking, and vertex-based distributed locking in Giraph. Each technique is an option that can be enabled and paired with Giraph async (AP) or GiraphUC (BAP) to provide serializability. We do not consider the constrained vertex-based solution for BSP (Proposition 2) as it further exacerbates BSP's already expensive communication and synchronization overheads (Chapter 3.1.1). We show that this does not impact usability in Chapter 4.4.6.

We use Giraph because it is a popular and performant system used, for example, by Facebook [25]. It is partition-aware and thus supports all three synchronization techniques. We do not implement token passing and partition-based locking in GraphLab async because, as described in Chapters 4.3.1.2 and 4.3.3.2, its architecture is optimized for vertex-based distributed locking and is not partition-aware. Adding partitions would require significant changes to GraphLab async, which is not the focus of this paper.

4.4.1 Giraph Background

As described in Chapter 4.3.1.2, Giraph assigns multiple graph partitions to each worker. During each superstep, each worker creates a pool of compute threads and pairs available threads with uncomputed partitions. Each worker maintains a message store to hold all incoming messages, while each compute thread uses a message buffer cache to batch outgoing messages to more efficiently utilize network resources. These buffer caches are automatically flushed when full but can also be flushed manually. In Giraph async and GiraphUC, messages between vertices of the same worker skip this cache and go directly to the message store.

Since Giraph is implemented in Java, it avoids garbage collection overheads (due to millions or billions of objects) by serializing vertex, edge, and message objects when not in use and deserializing them on demand. For each vertex u, Giraph stores only u's out-edges in u's vertex object. Thus, in-edges are not explicitly stored within Giraph.

4.4.2 Dual-Layer Token Passing

For dual-layer token passing, each worker uses three sets to track the vertex ids of local boundary, remote boundary, and mixed boundary vertices that it owns. p-internal vertices are determined by their absence from the three sets. We keep this type information separate from the vertex objects so that token passing is a modular option. Moreover, augmenting each vertex object with its type adds undesirable overheads since vertex objects must be serialized and deserialized many times throughout the computation. Having the type information in one place also allows us to update a vertex's type without deserializing its object.

To populate the sets, we intercept vertices during input loading and scan the partition ids of its out-edge neighbours to determine its type. This is sufficient for undirected graphs but not for directed graphs: a vertex u has no information about its in-edge neighbours. Thus, we have each vertex v send a message to its out-edge neighbours u that belong to a different partition. Then u can correct its type based on messages received from its in-edge neighbours. This all occurs during input loading and thus does not impact computation time. We also batch all dependency messages to minimize network overheads and input loading times.

As described in Chapter 4.3.2.2, the global and local tokens are passed around in a round-robin fashion. Local tokens are passed between a worker's partitions at the end of each superstep (logical superstep for BAP). Importantly, each worker holds the global token for n (logical) supersteps, where n is the number of partitions owned by that worker. This ensures that every partition's mixed boundary vertices are executed (Chapter 4.3.2.2). Without this, acquiring both tokens becomes a race condition, leading to starvation for some mixed boundary vertices.

Since local messages (between vertices of the same worker) are not cached, local replicas are updated eagerly. For remote replicas, each worker flushes and waits for delivery confirmations for its remote messages before passing along the global token. This enables message batching and is more performant than eagerly flushing after the computation of only a single vertex or partition.

In contrast, Giraphx [71] implements single-layer token passing and as a part of user algorithms rather than within the system. Giraphx stores type information with the vertex objects and uses two supersteps to update vertex types based on their in-edge dependencies. As discussed previously, the former is less performant while the latter wastes two supersteps of computation. Furthermore, this unnecessarily clutters user algorithms with system-level concerns and is thus neither transparent nor configurable.

4.4.3 Partition-based Distributed Locking

For partition-based distributed locking, each worker tracks fork and token states for its partitions in a dual-layer hash map. For each pair of neighbouring partitions P_i and P_j, we map P_i's partition id i to the id j to a byte whose bits indicate whether P_i has the fork, whether the fork is clean or dirty, and whether P_i holds the request token. Since partition ids are integers in Giraph, we use hash maps optimized for integer keys to minimize memory footprint.

In the Chandy-Misra algorithm, forks and tokens must be placed such that the precedence graph, whose edge directions determine which philosopher has priority for each shared fork, is initially acyclic [22]. A simple way to ensure this is to assign each philosopher an id and, for each pair of neighbours, give the token to the philosopher with the smaller id and the dirty fork to the one with the larger id. This guarantees that philosophers with smaller ids initially have precedence over all neighbours with larger ids, because a philosopher must give up a dirty fork upon request (except while it is eating). Partition ids naturally serve as philosopher ids, allowing us to use this initialization strategy.

For directed graphs, two neighbouring partitions may be connected by only a directed edge, due to their constituent vertices. Since partitions must be aware of both its in-edge and out-edge dependencies, workers exchange dependency information for their partitions during input loading. Like in token passing, dependency messages can be batched to ensure a minimal impact on input loading times.

Partitions acquire their forks synchronously by blocking until all forks arrive. This is because even if all forks are available, it takes time for them to arrive over the network, so immediately returning is wasteful and may prevent other partitions from executing (a partition cannot give up clean forks: it must first execute and dirty them). Finally, per Chapter 4.3.3, each worker flushes its remote messages before a partition sends a shared fork to another worker's partition.

4.4.4 Vertex-Based Distributed Locking

For vertex-based distributed locking, we use the insights from our implementation of partition-based locking. Each worker tracks fork and token states for its p-boundary vertices and uses vertex ids as keys. Keeping this data in a central per-worker data structure is even more important than in token passing: forks and tokens are constantly exchanged, so their states must be readily available to modify. Storing this data at each vertex object would incur significant deserialization overheads. Fork and token access patterns are also fairly random, which would further incur an expensive traversal of a byte array to locate the desired vertex.

Like the partition-based approach, for directed graphs, each vertex v broadcasts to its out-edge neighbours u so that u can record the in-edge dependency into the per-worker hash map. This occurs during input loading and all messages are batched. Vertices acquire their forks synchronously and each worker flushes its remote messages before any m-boundary vertex forfeits a fork to a vertex of another worker. However, these batches of remote messages are far too small to avoid significant communication overheads (Chapter 4.5.3).

In contrast, Giraphx sends forks and tokens as part of messages generated by user algorithms rather than as internal system messages. Consequently, fork and token messages between different workers are delivered only during global barriers, which unnecessarily divides each superstep into multiple sub-supersteps (akin to Proposition 2). The resulting increase in global barriers negatively impacts performance. Our implementation avoids this (Chapter 4.4.6). Furthermore, Giraphx again confuses system and algorithm concerns and is neither transparent nor configurable.

4.4.5 Fault Tolerance

For fault tolerance, the relevant data structures (hash sets or hash maps) are written to disk at a synchronous checkpoint. For token passing, each worker also records whether they have the global token and the id of the partition holding the local token. Checkpoints occur after a global barrier and thus capture a consistent state: there are no vertices executing and no in-flight messages. Thus, neither token passing's global token nor distributed locking's fork and request tokens are in transit.

4.4.6 Algorithmic Compatibility and Usability

A system can provide one computation model for algorithm developers to code with and execute algorithms using a different computation model. For example, Giraph async and GiraphUC are designed to allow algorithm developers to code their algorithms for BSP and transparently execute with the asynchronous AP and BAP models, respectively, to maximize performance. Thus, with respect to BSP, the AP and BAP models do not negatively impact usability.

When we pair Giraph async or GiraphUC with vertex-based or partition-based distributed locking, they remain backwards compatible with (i.e., can still execute) algorithms written for the BSP model. To take advantage of serializability, algorithm developers can now code for a serializable computation model. Specifically, this is the AP model with the additional guarantee that conditions C1 and C2 are true. For example, our graph coloring algorithm is written for this serializable AP model rather than for BSP (Chapter 4.5.2.1).

However, not all synchronization techniques provide this clean abstraction. Token passing fails in this regard because only a subset of vertices execute in each superstep.

That is, token passing cannot provide the guarantee that all vertices will execute some code in superstep i, because only a subset of the vertices will execute at superstep i. The same issue arises for the constrained vertex-based distributed locking solution for BSP and sync GAS (Proposition 2) and Giraphx's implementation of vertex-based distributed locking (Chapter 4.4.4), because they rely on global barriers for the exchange of forks and tokens. In contrast, our implementations of vertex-based and partition-based locking ensure that all vertices are executed exactly once in each superstep and thus provide superior compatibility and usability.

4.5 Experimental Evaluation

We compare dual-layer token passing and partition-based distributed locking with GiraphUC and vertex-based distributed locking with GraphLab async. We exclude Giraph async because it performs worse than GiraphUC for the same techniques. Similarly, GiraphUC is much slower than GraphLab async for vertex-based locking (Chapter 4.5.3). Hence, our evaluation focuses on the most performant combinations of systems and synchronization techniques.

Lastly, we exclude Giraphx as it implements token passing and vertex-based locking as part of user algorithms rather than within the system. This leads to poor performance and usability (Chapters 4.4.2 and 4.4.4). Furthermore, Giraphx uses an older and less performant version of Giraph and does not implement the more performant BAP model.

4.5.1 Experimental Setup

We evaluate the different synchronization techniques with the same setup used for our evaluation of GiraphUC (Chapter 3.4.1). However, we use 16 and 32 machines instead of 64 due to the poor performance of token passing and vertex-based distributed locking on the larger graphs (Chapter 4.5.3): using 64 machines can require over 8 hours for a single experimental run and does not reveal trends that are not already captured with 32 machines. We again implement our modifications in Giraph 1.1.0-RC0 and compare against GraphLab 2.2.

We continue to use large real-world datasets[5],[6][14, 16, 15], which are stored on HDFS as regular text files and loaded into each system using the default random hash partitioning.

[5] http://snap.stanford.edu/data/

[6] http://law.di.unimi.it/datasets.php

Table 4.1: Directed datasets. Parentheses give values for the undirected versions used by graph coloring.

Graph	$\|V\|$	$\|E\|$	**Max Degree**
com-Orkut **(OR)**	3.0M	117M (234M)	33K (33K)
arabic-2005 **(AR)**	22.7M	639M (1.11B)	575K (575K)
twitter-2010 **(TW)**	41.6M	1.46B (2.40B)	2.9M (2.9M)
uk-2007-05 **(UK)**	105M	3.73B (6.62B)	975K (975K)

Table 4.1 lists the four graphs we use: `AR`, `TW`, and `UK` are the same graphs from Table 3.1, while `OR` is a social network graph. We use `OR` as it is both larger and much denser than `US`. This makes it more suitable for testing algorithms such as graph coloring, which can fail to terminate on dense graphs when executed without serializability.

For partition-based distributed locking, we use Giraph's default setting of $|W|$ partitions per worker, where $|W|$ is the number of workers. Increasing the number of partitions beyond this does not improve performance: more edges become cut, which increases inter-partition dependencies and hence leads to more forks and tokens. Smaller partitions also mean smaller message batches and thus greater communication overheads. However, using too few partitions restricts parallelism for both compute threads and communication threads: the message store at each worker is indexed by separate hash maps for each partition, so more partitions enables more parallel modifications to the store while fewer partitions restricts parallelism and degrades performance.

4.5.2 Algorithms

We use graph coloring, PageRank, SSSP, and WCC as our algorithms. As discussed in Chapter 4.1, graph coloring requires serializability for termination. All algorithms have communication patterns that match those of more sophisticated algorithms that require serializability. For example, the ALS algorithm has identical communication patterns to PageRank: instead of performing scalar multiplication and addition, ALS has each vertex solve a linear system. By using simpler algorithms like PageRank, we can better understand the performance of the synchronization techniques without being hindered by the complexities of each algorithm. For example, because ALS requires matrix computations, we need to use a native matrix library for Giraph as it is implemented in Java: without such a library, matrix operations in Java can be up to 5× slower than in C++[7]. More

[7] It is for this reason that we do not use ALS: we were unable to find a native matrix library that would work in Giraph.

generally, to accurately parse the results of machine learning algorithms, we must factor in algorithm-specific concerns like training error. In contrast, for SSSP and WCC, we can simply look at computation time. Since our goal is to understand the performance of our synchronization techniques, rather than of particular algorithms, it suffices to use these four algorithms.

The PageRank, SSSP, and WCC algorithms we use are as described previously in Chapter 3.4.2. In particular, recall that GraphLab async has no notion of supersteps and can only terminate on PageRank based on a user-specified threshold ϵ (Chapter 3.4.2.4). To ensure that experiments complete in a reasonable amount of time, we terminate all systems using a threshold of 0.01 for `OR` and `AR` and 0.1 for `TW` and `UK`. This ensures that all systems perform the same amount of work for each graph and avoids the need to construct an exhaustive but expensive L_1-norm plot (Figure 3.7).

4.5.2.1 Graph Coloring

We use a greedy graph coloring algorithm (Algorithm 4) that requires serializability and an undirected input graph. Each vertex u initializes its value/color as NO_COLOR. Then, based on messages received from its (in-edge) neighbours, u selects the smallest non-conflicting color as its new color and broadcasts it to its (out-edge) neighbours.

Algorithm 4 Graph coloring pseudocode.

```
procedure COMPUTE(vertex, incoming messages)
    if superstep == 0 then
        vertex.setValue(NO_COLOR)
        return
    if vertex.getValue() == NO_COLOR then
        $c_{min}$ ← smallest non-conflicting color
        vertex.setValue($c_{min}$)
        Send $c_{min}$ to vertex's out-edge neighbours
    voteToHalt()
```

In theory, the algorithm requires only one iteration since serializability prevents conflicting colors. In practice, because Giraph async and GiraphUC are push-based, it requires three (logical) supersteps: initialization, color selection, and handling extraneous messages. The extraneous messages occur because vertices indiscriminately broadcast their current color, even to neighbours who are already complete. This wakes up vertices, leading to an additional iteration. GraphLab async, which is pull-based, has each vertex gather its neighbours' colors rather than broadcast its own and thus completes in a single iteration.

4.5.3 Results

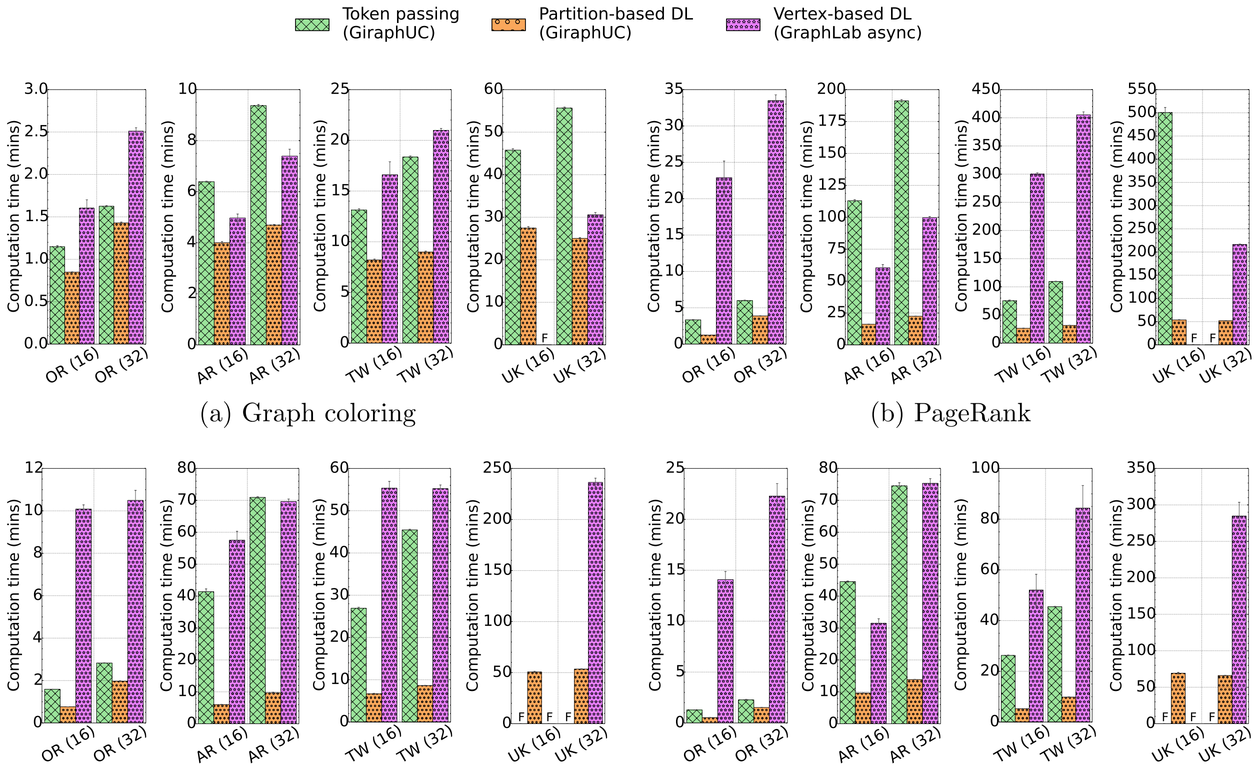
For our results, we report computation time, which is the total time of running an algorithm minus the input loading and output writing times. This also captures any communication overheads that the synchronization techniques may have: poor use of network resources translates to longer computation times. For each experiment, we report the mean and 95% confidence intervals of five runs (three runs for experiments taking over 3 hours).

As mentioned previously, we exclude Giraph async as it performs worse than GiraphUC for the same techniques. For example, on `OR`, token passing with Giraph async is up to 1.8× slower than GiraphUC. Similarly, on `OR`, vertex-based locking with GiraphUC is up to 44× slower than vertex-based locking with GraphLab async, as GraphLab async is tailored for this particular technique whereas GiraphUC is not (Chapter 4.3.1.2). Thus, we exclude these results and focus instead on the most performant combinations.

For graph coloring, partition-based locking is up to 2.3× faster than vertex-based locking for `TW` with 32 machines (Figure 4.3a). This is despite the fact that GiraphUC performs an additional iteration compared to GraphLab async (Chapter 4.5.2.1). Similarly, partition-based locking is up to 2.2× faster than token passing for `UK` on 32 machines. Vertex-based locking fails for `UK` on 16 machines because GraphLab async runs out of memory.

For PageRank, partition-based distributed locking again outperforms the other techniques: up to 18× faster than vertex-based locking on `OR` with 16 machines (Figure 4.3b). Vertex-based locking again fails for `UK` on 16 machines due to GraphLab async exhausting system memory. Token passing takes over 12 hours (720 mins) on `UK` when using 32 machines, meaning partition-based locking is over 14× faster than token passing.

For SSSP and WCC on `UK`, token passing takes over 7 hours (420 mins) for 16 machines and 9 hours (540 mins) for 32 machines, while GraphLab async fails on 16 machines due to running out of memory (Figures 4.3c and 4.3d). For SSSP, partition-based locking is up to 13× faster than vertex-based locking for `OR` on 16 machines and over 10× faster than token passing for `UK` with 32 machines. For WCC, partition-based locking is up to 26× faster than vertex-based locking for `OR` on 16 machines and over 8× faster than token passing for `UK` with 32 machines.



(a) Graph coloring

(b) PageRank

(c) SSSP

(d) WCC

Figure 4.3: Computation times for graph coloring, PageRank, SSSP, and WCC. Missing bars are labelled with ‘F’ for unsuccessful runs.

4.6 Summary

In this chapter, we presented a formalization of serializability for graph processing systems and proved that two key conditions are required to provide serializability. We introduced a novel partition-based synchronization technique to provide serializability and showed that, in addition to being correct, it is more efficient than existing techniques. We implemented all techniques in Giraph async and GiraphUC to provide serializability as a configurable option that is completely transparent to algorithm developers. Our experimental evaluation demonstrated that our partition-based technique is up to 26× faster than the existing approaches, even when compared to GraphLab async, a system specifically tailored for a vertex-based synchronization technique.

Chapter 5

Conclusion

We began by introducing two major problems of existing specialized Pregel-like graph processing systems: (1) poor performance due to frequent global synchronization barriers and (2) a lack of serializability guarantees for graph algorithms. Before addressing these two problems, we first categorized existing Pregel-like systems in Chapter 2, in part to illustrate that solutions to these two problems either do not exist or are inadequate. We also characterized several other related systems in the field of graph processing, to show both how the field has evolved and the future directions in which it may be heading.

In Chapter 3, we addressed problem (1) by introducing a new barrierless asynchronous parallel (BAP) computation model that uses the notions of local barriers and logical supersteps to reduce the frequency of global barriers while maintaining correctness for algorithms written for BSP. Consequently, this allows a system to provide algorithm developers with a BSP interface while transparently executing using the asynchronous BAP model to maximize performance. Our implementation of the BAP model, GiraphUC, demonstrated across-the-board performance gains of up to 5× faster than Giraph, Giraph async, and GraphLab sync and 86× faster than GraphLab async.

In Chapter 4, we addressed problem (2) by formalizing the notion of serializability for Pregel-like systems and proving that systems can provide serializability when paired with a synchronization technique that satisfies two key conditions. We then described and characterized the performance of existing synchronization techniques (token passing and vertex-based distributed locking) and introduced a new and more performant partition-based distributed locking technique. Our implementations of these techniques in Giraph async and GiraphUC demonstrated that our partition-based distributed locking technique outperforms existing techniques by up to 26×, even when compared to GraphLab async, a system tailored for the vertex-based distributed locking technique.

5.1 Future Work

There remains several interesting research questions for both GiraphUC and serializability. Specifically, how to provide support for aggregators that require global coordination in GiraphUC (Chapter 5.1.1) and how to generalize serializability to algorithms where vertices can communicate with arbitrary vertices (Chapter 5.1.2). We also pose broader long-term questions in Chapter 5.1.3 regarding the integration of BAP and serializability into generic big data systems.

5.1.1 Globally Coordinated Aggregators

In BSP, algorithms use aggregators to record global information: for example, the number of vertices that successfully completed some task in the previous superstep. Each vertex can contribute a value to an aggregator in superstep i and that aggregator's value is made available to all vertices in superstep $i + 1$. However, in many cases, algorithms require only per-worker information for correctness. That is, they do not require aggregators that are globally coordinated: instead of each vertex needing to see an aggregated value accumulated from all vertices of all workers in a previous superstep, a vertex of worker W_j needs to see only the aggregated value accumulated from W_j's vertices in the previous logical superstep.

GiraphUC supports aggregators that do not need to be globally coordinated since, in practice, few algorithms require globally coordinated aggregators. For example, in Table 2.1, only K-means requires globally coordinated aggregators. Additionally, since GiraphUC supports multi-phase algorithms, it naturally supports globally coordinated aggregators used for denoting which computation phase the algorithm is in. However, GiraphUC does not efficiently support other forms of globally coordinated aggregators: ensuring that the aggregator's value is always correct requires all workers to wait for each other, which means there must be frequent global barriers and so BAP effectively degrades to AP.

Thus, an open question is whether it is possible to support globally coordinated aggregators in a more efficient manner, while maintaining correctness under the BSP model. For example, instead of synchronizing all workers with a global barrier, can we have faster workers use stale aggregator values in lieu of receiving newer values from their slower neighbours (similar to Theorem 2)? Or, alternatively, can globally coordinated aggregators be converted into equivalent per-worker aggregators? In the latter case, the BSP interface can be modified slightly, as done for multi-phase algorithms (Chapter 3.2.3), to allow algorithm developers to specify these per-worker aggregators.

5.1.2 Generalized Serializability

In Chapter 4, we formalized serializability for algorithms where vertices communicate only with their direct neighbours. This covers many machine learning and graph analytic algorithms that require serializability. In fact, we were unable to find any examples of algorithms that required both serializability and communication with arbitrary vertices.

Nevertheless, it would be useful to generalize serializability support to include algorithms that communicate with arbitrary vertices. Supporting this within the graph processing system is much more challenging as the read set of each transaction is no longer known a priori (Chapter 4.2.2), meaning condition C2 of Theorem 4 is insufficient for guaranteeing non-conflicting operations. That is, the read set of each vertex is, in general, determined dynamically at runtime.

One possible solution is to provide an API that, for each vertex u, enables the algorithm developer to instruct the system on which vertices u must be coordinated with prior to u being executed. However, this may not always be possible if, for example, within a single superstep, each vertex u sends messages to vertices whose ids are given by the received messages that u must first read. To get around this, one might further modify the interface to separate message reading from the compute function. However, such a solution will negatively impact usability and compatibility.

A more general solution is to introduce the notion of an abort, to allow for transactions to be cancelled and rolled back. However, this requires a significant and non-trivial redesign of the graph processing system to make it transaction-aware: it must track and explicitly commit or abort all transactions. In contrast, the synchronization techniques we introduced in Chapter 4.3 do not require the system to be transaction-aware: the techniques simply run on top of the existing system. A redesign is necessary to support aborts because systems that are not transaction-aware will suffer cascading aborts: messages will be erroneously sent before a commit occurs and there is no simple way to undo a message once it is sent. In the worst case, such a scenario would require rolling back the entire graph state.

5.1.3 Asynchronous Models in Generic Systems

There is now a greater focus on providing graph analytics on generic big data systems such as dataflow engines, like Spark [82] and Hyracks [18], and relational databases (Chapter 2.4). Unlike specialized graph processing systems, where there is a direct mapping from graph analytics to the internal system implementation, graph analytics built on generic big data systems are abstracted away from the underlying system implementation. For example, dataflow engines simulate graph operations by performing joins on tables while relational databases translate graph operations to relational operations.

Thus, a more general open question is whether asynchronous models, such as BAP, can be implemented on these generic big data systems to provide performance gains despite the extra layer of abstraction. Alternatively, can the notions of local barriers and logical supersteps be translated into mechanisms that affect the implementation of generic big data systems to enhance their graph analytics performance?

Similarly, it may be possible to continue to provide serializability in a configurable and transparent manner by translating the synchronization techniques for use with these generic big data systems. In particular, relational databases may be a better option for providing generalized serializability (Chapter 5.1.2), as they are typically transaction-aware and can thus avoid the non-trivial redesign required by specialized graph processing systems.

APPENDICES

Appendix A

Proofs for Giraph Unchained

In this chapter, we provide proofs for Theorems 1 and 2 of Chapter 3.

A.1 Preliminaries

Single-phase (BSP) algorithms are so named because they have a single computation phase. That is, the compute function executed by vertices always contains the same logic for every superstep. For example, if the compute function contains branching logic based on the superstep count or some aggregator value, then it is composed of multiple computation phases[1] and therefore not a single-phase algorithm. The exceptions to this are initialization, which is conditional on it being the first superstep, and termination, such as voting to halt after K supersteps. The former is a special case that we can handle directly in the implementation, while the latter is termination and so there is no additional logic that will follow. Specifically, many algorithms perform initialization procedures in the first (logical) superstep and send, but do not receive, messages. This can be handled by keeping all messages in the message store until the second (logical) superstep. This is correct, since in BSP no sent messages will be seen by any vertex in the first (logical) superstep.

Since both the AP and BAP models execute only algorithms implemented for the BSP model, we need only consider single-phase BSP algorithms. From the discussion above, such algorithms have the following property:

[1] Theorem 3 describes how the AP and BAP models handle multi-phase algorithms.

Property I: The computation logic for single-phase BSP algorithms is the same for every superstep.

Secondly, because of the iterative nature of BSP, a message m sent by an arbitrary vertex v is either a function of v's vertex value or it is not. That is, an algorithm either has all messages of the form $m = f(v_{\text{value}})$, where v is the sender of m, or all messages of the form $m \neq f(v_{\text{value}})$. All messages of any particular algorithm are always of one form because, by Property I, the computation logic generating the messages is always the same. In the first case, v's vertex value is a function of v's previous vertex values, so m encapsulates v's most recent vertex value as well as all past values, meaning that newer messages contain more information than older messages. Then, since newer messages are more important than older messages, each vertex needs to see only the newest messages sent to it. In the second case, the algorithm is an accumulative update algorithm [85] where old messages are as important as new messages, because m does not encapsulate v's previous values. Hence, each vertex must see all messages sent to it exactly once. Then we have the following two properties:

Property II: For an algorithm with messages $m = f(v_{\text{value}})$, where m is sent by a vertex v, a new message from v to a vertex u is more important than an old message from v to u and so u must see the newest message sent to it by v.

Property III: For an algorithm with messages $m \neq f(v_{\text{value}})$, where m is sent by a vertex v, all messages from v to a vertex u are important and so u must see all messages sent to it by v exactly once.

Note that both Properties II and III are enforced in the BSP model since implementations ensure that messages are delivered exactly once and not lost, by acknowledging delivery of messages and retrying failed sends. Since message stores buffer all received messages exactly once and remove all the messages for a vertex u when computing u, the vertex u will see all messages sent to it exactly once.

All single-phase BSP algorithms can be defined as one of two types: ones in which vertices do not need all messages from all neighbours (**type A**) and ones in which vertices need all messages from all neighbours (**type B**). Since accumulative update algorithms are always type A algorithms [85], messages of type A algorithms are either all of the form $m = f(v_{\text{value}})$ or all of the form $m \neq f(v_{\text{value}})$, while messages of type B algorithms are always of the form $m = f(v_{\text{value}})$.

A.1.1 Type A

Type A algorithms satisfy both Properties II and III above, as it must handle messages of both forms. Since Property III is more restrictive (that is, it being true implies Property II is also true), the invariant for type A algorithms is simply Property III:

Invariant A: All messages sent from a vertex v to a vertex u must be seen by u exactly once.

Furthermore, the definition of type A algorithms places no constraints on when a vertex's neighbours are active or inactive. That is, a vertex's neighbours can become active or inactive at any superstep during the computation. Consequently, its compute function can receive any number of messages in each superstep, which means it must be capable of processing such messages (otherwise, the algorithm would not be correct even for BSP):

Property A: The compute function of type A algorithms correctly handles any number of messages in each superstep.

A.1.2 Type B

Unlike type A algorithms, type B algorithms have only one form of messages so it satisfies only Property II. However, since the definition of a type B algorithm is more restrictive than Property II, its invariant follows directly from its definition:

Invariant B: Each vertex must see exactly one message from each of its in-edge neighbours at every superstep.

For type B algorithms, a vertex's neighbours are always active since each vertex must receive messages from all their neighbours. Intuitively, if even one vertex is inactive, it will stop sending messages to its neighbours and thus violate the definition of type B algorithms. Since all messages for type B algorithms are of the form $m = f(v_{\text{value}})$, then by Property II, not every message from a vertex v to another vertex u is important because we will always see a newer message from v (unless the algorithm is terminating, but in that case u would be halting as well). That is, correctness is still maintained when a vertex u sees only a newer message m' sent to it by v and not any older messages m from v, since m' is more important than m. Therefore:

Property B: An old message m from vertex v to u can be overwritten by a new message m' from v to u.

Note that, unlike type A algorithms, the compute function of type B algorithms can only correctly process N messages when executing on a vertex u with N in-edge neighbours. Processing fewer or more messages will cause vertex u to have an incorrect vertex value. Invariant B ensures that this constraint on the computation logic is satisfied.

A.2 Theorem 1

Theorem 1. *The AP and BAP models correctly execute single-phase BSP algorithms in which vertices do not need all messages from all neighbours.*

Proof. As mentioned in Appendix A.1 above, we need only focus on the type A algorithms. Specifically, we must show that (1) relaxing message isolation and, for BAP, removing global barriers does not impact the algorithm's correctness and (2) Invariant A is enforced by both models.

For (1), relaxing message isolation means that messages are seen earlier than they would have been seen in the BSP model, while the removal of global barriers means a message from any superstep can arrive at any time.

By Property I, a received message will be processed in the same way in the current superstep as it would in any other superstep, because the computation logic is the same in all supersteps. Thus, showing messages from different supersteps does not affect correctness, meaning that relaxing message isolation does not impact correctness.

By Property A, vertices can process any number of received messages. That is, correctness is maintained even if messages are delayed[2] and hence not passed to the vertex's compute function or a large number of messages suddenly arrive and are all passed to the vertex's compute function. Together with Property I, this shows that removing global barriers does not impact correctness.

For (2), like BSP, the message stores of both the AP and BAP models still buffer all received messages by default and, when computing a vertex u, will remove all messages for u from the message store (Chapter 3.3.2). Since the implementations of both models ensure messages are delivered exactly once and are not lost, we can ensure that the message store buffers all messages for u exactly once and thus u sees all messages from senders v exactly once. More precisely, the AP and BAP models still guarantee message delivery like

[2] At the implementation level, messages are guaranteed to be delivered, so a message may be delayed but is never lost.

the BSP model—the models modify *when* a message is received, not *whether* a message will be delivered. This is enforced in the implementation in the same way as it is for implementations of the BSP model: by using acknowledgements and resending messages that fail to deliver. □

A.3 Theorem 2

Theorem 2. *Given a message store that is initially filled with valid messages, retains old messages, and overwrites old messages with new messages, the BAP model correctly executes single-phase BSP algorithms in which vertices need all messages from all neighbours.*

Proof. As mentioned in Appendix A.1, we need only focus on the type B algorithms. Specifically, we must show that (1) relaxing message isolation and removing global barriers does not impact the algorithm's correctness and (2) Invariant B is maintained by the BAP model. Note that proving Theorem 2 for the BAP model also proves it for the AP model since BAP subsumes AP.

First, note that the theorem states three assumptions about the message store: it is initially filled with valid messages, it retains old messages (i.e., does not remove them), and new messages overwrite old messages. More concretely, consider an arbitrary vertex u with neighbours v_i, $0 \leq i < N$, where $N = \deg^{-}(u)$ is the number of in-edges (or in-edge neighbours) of u. Then, initially, the message store will have exactly one message m_i from each of u's neighbours v_i. That is, when computing u, the system will pass the set of messages $S = \{m_i\}_{\forall i}$, with $|S| = N$, to u's compute function. If u is to be computed again and no new messages have arrived, the message store retains all the old messages and so $S = \{m_i\}_{\forall i}$ is simply passed to u's compute function again. If a new message m'_j arrives from one of u's neighbours v_j, $j \in [0, N)$, it will overwrite the old message m_j. From Property B, performing message overwrites in this way will not affect correctness. Then the set of messages passed to u's compute function will now be $S = \{m_i\}_{\forall i} \cup \{m'_j\} \setminus \{m_j\}$, where again we preserve $|S| = N$ and exactly one message from each of u's in-edge neighbours.

Then by the above, the BAP model maintains Invariant B: on every logical superstep, the message store passes to u exactly one message from each of its in-edge neighbours. Hence, (2) is true. For (1), relaxing message isolation does not affect correctness since, by Property I, received messages are processed in the same way in every superstep and, since (2) is true, we already satisfy Invariant B. Removing global barriers also does not affect correctness since messages can be delayed or arrive in large numbers without affecting the number or source of messages in the message store: old messages are retained if messages are delayed, while newly arrived messages overwrite existing ones. Thus, the message store will still have exactly one message from each of u's in-edge neighbours. □

References

[1] Apache Giraph. http://giraph.apache.org.

[2] Apache Hadoop. http://hadoop.apache.org.

[3] Apache Hama. http://hama.apache.org.

[4] Neo4j. http://www.neo4j.com.

[5] Okapi. http://grafos.ml/okapi.

[6] OrientDB. http://www.orientdb.com.

[7] Titan. http://thinkaurelius.github.io/titan/.

[8] GraphLab: Distributed Graph-Parallel API. http://docs.graphlab.org/classgraphlab_1_1async__consistent__engine.html, 2014.

[9] Lada A. Adamic and Eytan Adar. Friends and neighbors on the web. *Social Networks*, 25(3):211–230, 2003.

[10] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinlnder, Matthias J. Sax, Sebastian Schelter, Mareike Hger, Kostas Tzoumas, and Daniel Warneke. The Stratosphere Platform for Big Data Analytics. *The VLDB Journal*, 23(6):939–964, 2014.

[11] Lars Backstrom, Paolo Boldi, Marco Rosa, Johan Ugander, and Sebastiano Vigna. Four Degrees of Separation. In *WebSci '12*, pages 33–42, 2012.

[12] Nguyen Thien Bao and Toyotaro Suzumura. Towards Highly Scalable Pregel-based Graph Processing Platform with x10. In *WWW '13 Companion*, pages 501–508, 2013.

[13] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, 1987.

[14] P. Boldi and S. Vigna. The Webgraph Framework I: Compression Techniques. In *WWW '04*, pages 595–602, 2004.

[15] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. UbiCrawler: A Scalable Fully Distributed Web Crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[16] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *WWW '11*, pages 587–596, 2011.

[17] Paolo Boldi and Sebastiano Vigna. Four Degrees of Separation, Really. http://arxiv.org/abs/1205.5509, 2012.

[18] Vinayak R. Borkar, Michael J. Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing. In *ICDE '11*, pages 1151–1162, 2011.

[19] Joseph K. Bradley, Aapo Kyrola, Danny Bickson, and Carlos Guestrin. Parallel Coordinate Descent for L1-Regularized Loss Minimization. In *ICML*, 2011.

[20] Yingyi Bu, Vinayak Borkar, Jianfeng Jia, Michael J. Carey, and Tyson Condie. Pregelix: Big(ger) Graph Analytics on A Dataflow Engine. *PVLDB*, 8(2):161–172, 2015.

[21] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *PVLDB*, 3(1–2):285–296, 2010.

[22] K. M. Chandy and J. Misra. The Drinking Philosophers Problem. *ACM Trans. Program. Lang. Syst.*, 6(4):632–646, 1984.

[23] Rishan Chen, Xuetian Weng, Bingsheng He, and Mao Yang. Large Graph Processing in the Cloud. In *SIGMOD '10*, pages 1123–1126, 2010.

[24] Jiefeng Cheng, Qin Liu, Zhenguo Li, Wei Fan, John C.S. Lui, and Cheng He. VENUS: Vertex-Centric Streamlined Graph Computation on a Single PC. In *ICDE '15*, 2015.

[25] Avery Ching. Scaling Apache Giraph to a trillion edges. http://www.facebook.com/10151617006153920, 2013.

[26] Sun Chung and Anne Condon. Parallel Implementation of Borvka's Minimum Spanning Tree Algorithm. In *IPPS '96*, pages 302–308, 1996.

[27] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[28] Gianmarco De Francisci Morales, Aristides Gionis, and Mauro Sozio. Social Content Matching in MapReduce. *Proceedings of the VLDB Endowment*, 4(7):460–469, 2011.

[29] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI '04*, 2004.

[30] Jim Edwards. 'Facebook Inc.' Actually Has 2.2 Billion Users Now. http://www.businessinsider.com/facebook-inc-has-22-billion-users-2014-7, 2014.

[31] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: A Runtime for Iterative MapReduce. In *HPDC '10*, pages 810–818, 2010.

[32] Jing Fan, Adalbert Gerald, Soosai Raj, and Jignesh M. Patel. The Case Against Specialized Graph Analytics Engines. In *CIDR '15*, 2015.

[33] Joseph Gonzalez, Yucheng Low, Arthur Gretton, and Carlos Guestrin. Parallel Gibbs Sampling: From Colored Fields to Thin Junction Trees. In *AISTATS*, volume 15, pages 324–332, 2011.

[34] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI '12*, pages 17–30, 2012.

[35] Google. How search works. http://www.google.com/insidesearch/howsearchworks/thestory/, 2014.

[36] Yong Guo, Marcin Biczak, Ana Lucia Varbanescu, Alexandru Iosup, Claudio Martella, and Theodore L. Willke. How Well do Graph-Processing Platforms Perform? An Empirical Performance Evaluation and Analysis. In *IPDPS 2014*, 2014.

[37] M.N. Gurcan, L.E. Boucheron, A. Can, A. Madabhushi, N.M. Rajpoot, and B. Yener. Histopathological Image Analysis: A Review. *IEEE Rev Biomed Eng*, 2:147–171, 2009.

[38] Minyang Han and Khuzaima Daudjee. Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Processing Systems. *PVLDB*, 8(9):950–961, 2015.

[39] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M. Tamer Özsu, Xingfang Wang, and Tianqi Jin. An Experimental Comparison of Pregel-like Graph Processing Systems. *PVLDB*, 7(12):1047–1058, 2014.

[40] 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 '13*, pages 77–85, 2013.

[41] Imranul Hoque and Indranil Gupta. LFGraph: Simple and Fast Distributed Graph Analytics. In *TRIOS '13*, pages 9:1–9:17, 2013.

[42] Alekh Jindal, Praynaa Rawlani, Eugene Wu, Samuel Madden, Amol Deshpande, and Mike Stonebraker. Vertexica: Your Relational Friend for Graph Analytics! *PVLDB*, 7(13):1669–1672, 2014.

[43] U. Kang, Charalampos E. Tsourakakis, Ana Paula Appel, Christos Faloutsos, and Jure Leskovec. HADI: Mining Radii of Large Graphs. *ACM TKDD*, 5(2):8:1–8:24, 2011.

[44] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In *ICDM '09*, pages 229–238, 2009.

[45] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: A system for dynamic load balancing in large-scale graph processing. In *EuroSys '13*, pages 169–182, 2013.

[46] Yehuda Koren. Factorization Meets the Neighborhood: A Multifaceted Collaborative Filtering Model. In *KDD '08*, pages 426–434, 2008.

[47] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix Factorization Techniques for Recommender Systems. *IEEE Computer*, 42(8):30–37, 2009.

[48] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale Graph Computation on Just a PC. In *OSDI '12*, pages 31–46, 2012.

[49] Ian X. Y. Leung, Pan Hui, Pietro Liò, and Jon Crowcroft. Towards real-time community detection in large networks. *Physical Review E*, 79(6):066107, 2009.

[50] 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. *PVLDB*, 5(8):716–727, 2012.

[51] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. Large-scale distributed graph computing systems: An experimental evaluation. *Proceedings of the VLDB Endowment*, 8(3), 2014.

[52] Michael Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem. *SIAM Journal on Computing*, 15(4):1036–1053, 1986.

[53] Maja Kabiljo. Improve the way we keep outgoing messages. http://issues.apache.org/jira/browse/GIRAPH-388, 2012.

[54] Grzegorz Malewicz, Matthew H. Austern, Aart JC Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD/PODS '10*, pages 135–146, 2010.

[55] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking Like a Vertex: a Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing. 2015.

[56] Simona Mihaela Orzan. *On Distributed Verification and Verified Distribution*. PhD thesis, Free University of Amsterdam, 2004.

[57] Robert Preis. Linear Time 1/2-Approximation Algorithm for Maximum Weighted Matching in General Graphs. In *STACS '99*, pages 259–269, 1998.

[58] Natasa Przulj. Protein-protein interactions: Making sense of networks via graph-theoretic modeling. *BioEssays*, 33(2):115–123, 2011.

[59] Louise Quick, Paul Wilkinson, and David Hardcastle. Using Pregel-like Large Scale Graph Processing Frameworks for Social Network Analysis. In *ASONAM '12*, pages 457–463, 2012.

[60] Usha Nandini Raghavan, Reka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3):036106, 2007.

[61] Marko A. Rodriguez. Faunus Provides Big Graph Data Analytics. http://thinkaurelius.com/2012/11/11/faunus-provides-big-graph-data-analytics/, 2012.

[62] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: Edge-centric Graph Processing Using Streaming Partitions. In *SOSP '13*, pages 472–488, 2013.

[63] Semih Salihoglu and Jennifer Widom. GPS: A Graph Processing System. In *SSDBM '13*, pages 22:1–22:12, 2013.

[64] Semih Salihoglu and Jennifer Widom. Optimizing Graph Algorithms on Pregel-like Systems. Technical report, Stanford, 2013.

[65] Zechao Shang and Jeffrey Xu Yu. Catch the Wind: Graph Workload Balancing on Cloud. In *ICDE '13*, pages 553–564, 2013.

[66] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *SIGMOD '13*, pages 505–516, 2013.

[67] Athanassios G. Siapas. *Criticality and Parallelism in Combinatorial Optimization.* PhD thesis, Massachusetts Institute of Technology, 1996.

[68] David Simmen, Karl Schnaitter, Jeff Davis, Yingjie He, Sangeet Lohariwala, Ajay Mysore, Vinayak Shenoi, Mingfeng Tan, and Yu Xiao. Large-scale Graph Analytics in Aster 6: Bringing Context to Big Data Discovery. *PVLDB*, 7(13):1405–1416, 2014.

[69] Philip Stutz, Abraham Bernstein, and William Cohen. Signal/Collect: Graph Algorithms for the (Semantic) Web. In *ISWC'10*, pages 764–780, 2010.

[70] Siddharth Suri and Sergei Vassilvitskii. Counting Triangles and the Curse of the Last Reducer. *WWW '11*, pages 607–614, 2011.

[71] Serafettin Tasci and Murat Demirbas. Giraphx: Parallel Yet Serializable Large-scale Graph Processing. In *Euro-Par '13*, pages 458–469, 2013.

[72] 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.

[73] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33(8):103–111, 1990.

[74] Guozhang Wang, Wenlei Xie, Alan J. Demers, and Johannes Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR '13*, 2013.

[75] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. SYNC or ASYNC: Time to Fuse for Distributed Graph-parallel Computation. In *PPoPP'15*, 2015.

[76] Wenlei Xie, Guozhang Wang, David Bindel, Alan Demers, and Johannes Gehrke. Fast Iterative Graph Computation with Block Updates. *PVLDB*, 6(14):2014–2025, 2013.

[77] Reynold S. Xin, Daniel Crankshaw, Ankur Dave, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. GraphX: Unifying Data-Parallel and Graph-Parallel Analytics. http://arxiv.org/abs/1402.2394, 2014.

[78] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. GraphX: A Resilient Distributed Graph System on Spark. In *GRADES '13*, pages 2:1–2:6, 2013.

[79] Jilong Xue, Zhi Yang, Zhi Qu, Shian Hou, and Yafei Dai. Seraph: An Efficient, Low-cost System for Concurrent Graph Processing. In *HPDC '14*, pages 227–238, 2014.

[80] Da Yan, James Cheng, Yi Lu, and Wilfrend Ng. Blogel: A Block-Centric Framework for Distributed Computation on Real-World Graphs. *PVLDB*, 7(14):1981–1992, 2014.

[81] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. Pregel Algorithms for Graph Connectivity Problems with Performance Guarantees. *Proceedings of the VLDB Endowment*, 7(14):1821–1832, 2014.

[82] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *HotCloud '10*, 2010.

[83] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. iMapReduce: A Distributed Computing Framework For Iterative Computation. In *DataCloud '11*, pages 1112–1121, 2011.

[84] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. PrIter: A Distributed Framework for Prioritized Iterative Computations. In *SOCC '11*, pages 13:1–13:14, 2011.

[85] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Accelerate Large-scale Iterative Computation Through Asynchronous Accumulative Updates. In *ScienceCloud '12*, pages 13–22, 2012.

[86] 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'15*, pages 45–58, 2015.

[87] Chang Zhou, Jun Gao, Binbin Sun, and Jeffrey Xu Yu. MOCgraph: Scalable Distributed Graph Processing Using Message Online Computing. *PVLDB*, 8(4):377–388, 2014.

[88] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-Scale Parallel Collaborative Filtering for the Netflix Prize. In *AAIM '08*, pages 337–348, 2008.