Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Processing Systems

University of Waterloo Technical Report CS-2015-04*

Minyang Han
David R. Cheriton School of Computer Science
University of Waterloo
m25han@uwaterloo.ca

Khuzaima Daudjee
David R. Cheriton School of Computer Science
University of Waterloo
kdaudjee@uwaterloo.ca

ABSTRACT

The bulk synchronous parallel (BSP) model used by synchronous graph processing systems allows algorithms to be easily implemented and reasoned about. However, BSP can suffer from poor performance due to stale messages and frequent global synchronization barriers. Asynchronous computation models have been proposed to alleviate these overheads but existing asynchronous 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 reduces both message staleness and global synchronization. This enables BAP to overcome the limitations of existing asynchronous models 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 our system at scale with large real-world graphs on 64 EC2 machines. We show that GiraphUC provides across-the-board performance improvements of up to 5× faster over synchronous systems and up to an order of magnitude faster than asynchronous systems. Our results demonstrate that the BAP model provides efficient and transparent asynchronous execution of algorithms that are programmed synchronously.

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 [17]. 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 [15]. Scientists are also leveraging biology graphs to understand protein interactions [27] and cell graphs for automated cancer diagnosis [18].

*This technical report is the extended version of a paper published in PVLDB 2015 [19].

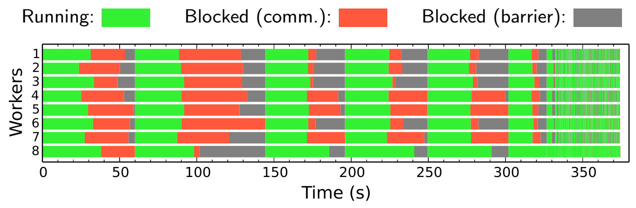


Figure 1: Communication and synchronization overheads for a BSP execution of weakly connected components using 8 worker machines on `TW` (Table 1).

These graph processing problems are solved by implementing and running specific graph algorithms on input graphs of interest. *Graph processing systems* provide an API for developers to implement such algorithms and a means to run these algorithms against desired input graphs. Google's Pregel [26] is one such system that provides a native graph processing API by pairing the bulk synchronous parallel (BSP) computation model [35] with a vertex-centric, or "think like a vertex", programming model. This has inspired popular open source Pregel-like graph processing systems such as Apache Giraph [1] and GraphLab [24].

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 [24]. 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. For *push-based* systems such as Giraph [1], Giraph++ [34], and GRACE [36], 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 scalability [20].

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 illustrates an actual BSP execution of the weakly connected components algorithm in which workers are, on aver-

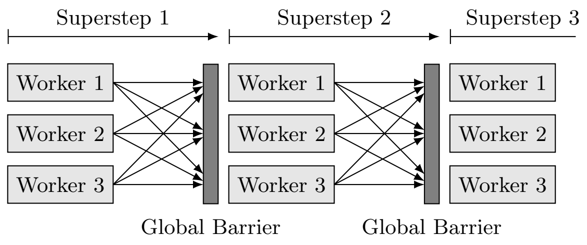


Figure 2: The BSP model, illustrated with three supersteps and three workers [22].

age, blocked on communication and synchronization for 46% of the total computation time. GraphLab's asynchronous mode 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 [36], impeding usability, and do not support algorithms that require graph mutations [24, 36] or algorithms with multiple computation phases [34], 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 asynchronous GraphLab and achieves good performance improvements over synchronous Giraph, asynchronous Giraph (which uses the AP model), and synchronous and asynchronous GraphLab. Thus, GiraphUC enables developers to code for a synchronous BSP model and transparently execute with an asynchronous BAP model to maximize performance.

Our contributions are hence threefold: **(1)** we show that the BAP model greatly reduces the frequency of global barriers without the use of distributed locking; **(2)** we implement the BAP model in Giraph to demonstrate that it provides across-the-board performance gains of up to 5× over synchronous systems and up to 86× over asynchronous systems; and **(3)** at the user level, we show that the BAP model provides a "program synchronously, execute asynchronously" paradigm, *transparently* providing performance gains without sacrificing usability or compatibility.

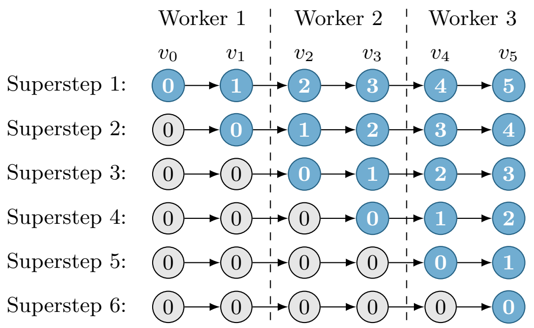


Figure 3: BSP execution of a WCC example. Gray vertices are inactive. Blue vertices have updated vertex values.

This paper is organized as follows. In Section 2, we provide background on the BSP, AP, and GAS models and their limitations. In Section 3, we describe our BAP model and in Section 4, its implementation in Giraph, GiraphUC. We provide an extensive experimental evaluation of GiraphUC in Section 5 and describe related work in Section 6 before concluding in Section 7.

2. BACKGROUND AND MOTIVATION

We first introduce the BSP model used by Pregel and Giraph, followed by the AP model, an asynchronous extension of BSP. We then consider the GAS model used by GraphLab as a point of comparison before motivating the BAP model by discussing the limitations of BSP and AP.

2.1 The BSP Model

Bulk synchronous parallel (BSP) [35] is a parallel computation model in which computations are divided into a series of (BSP) *supersteps* separated by global barriers (Figure 2). 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 3). In the first superstep, a 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 (Section 5.2.2). 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, 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.

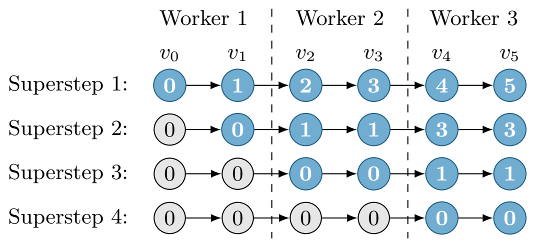


Figure 4: AP execution of the WCC example.

2.2 The 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 (Section 4.2).

To see how reducing staleness can improve performance, consider again our WCC example from Figure 3. For simplicity, assume that workers are single-threaded so that they execute their two vertices sequentially. Then, in the BSP execution (Figure 3), 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 4), 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 (Section 2.4).

2.3 Comparison to the 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 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. GraphLab, like Pregel, pairs GAS with a vertex-centric programming model. However, as evidenced by the Gather phase, GAS and GraphLab are pull-based rather than push-based.

For synchronous execution, GraphLab sync queues all vertices to be executed into a scheduler. During each iteration, or superstep, all vertices collectively execute Gather, then Apply, then Scatter. Hence, like BSP, vertices can use information from only the previous superstep. Vertices that are activated during Scatter are placed back into the scheduler queue to be executed again in the next superstep. This is different from BSP, where vertices are active by default and must choose to be inactive. Overall, the synchronous execution of GAS is quite similar to BSP. For example, WCC would proceed similarly to Figure 3.

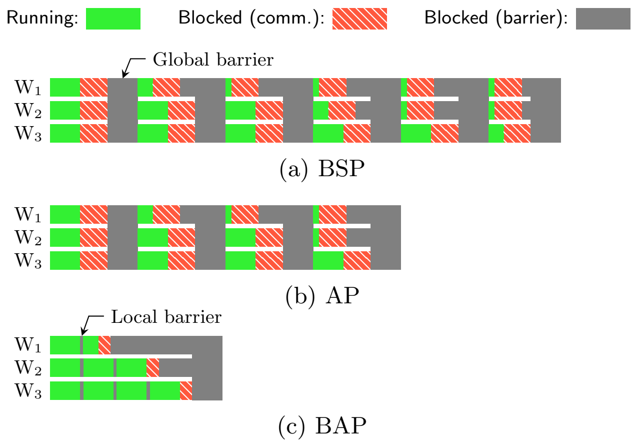


Figure 5: Computation times for the WCC example under different computation models.

For asynchronous execution, GraphLab async maintains a large pool of lightweight threads (called fibers), each of which independently executes all three GAS phases on any scheduled vertex. Unlike synchronous execution, this enables vertices to pull the most recent information from their neighbours and allows reactivated vertices to be immediately executed again. To ensure consistency, GraphLab async uses distributed locking to acquire a write lock on the executing vertex and read locks on the vertex's neighbours [4]. The locks ensure that the executing vertex does not pull inconsistent data from its neighbours. To terminate, GraphLab async runs a distributed consensus algorithm to check that all workers' schedulers are empty.

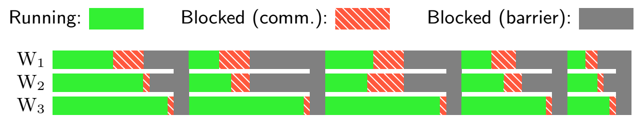
In contrast to asynchronous GAS, AP is push-based and can avoid expensive distributed locking because messages are received only after a vertex finishes its computation and explicitly pushes such messages. Since messages are buffered in a local message store, concurrent reads and writes to the store can be handled locally with local locks or lock-free data structures. Additionally, asynchronous GAS has no notion of supersteps and uses a distributed consensus algorithm to check for termination, whereas the AP model has supersteps and relies on the master to check for termination.

2.4 Limitations and Opportunities

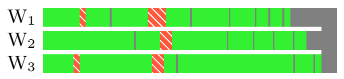
2.4.1 *Performance*

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 5a and 5b visualize the BSP and AP executions of our WCC example 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) BAP

Figure 6: WCC computation times based on real executions of 16 workers on TW (Table 1).

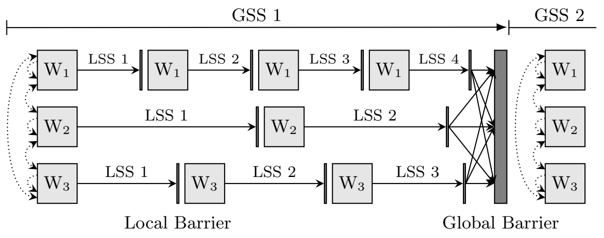


Figure 7: The BAP model, with two global supersteps and three workers. GSS stands for global superstep, while LSS stands for logical superstep.

Stale messages. As described in Section 2.2, 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 [24]. Our proposed BAP model preserves these advantages of the AP model by also reducing message staleness without using distributed locking (Section 3).

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 5a and 5b, 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 5c 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 (Section 3), 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 small number of vertices [33]. 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 6, which is based on real executions of a large real-world graph. In the BSP execution (Figure 6a), 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 6b) 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.

2.4.2 *Algorithmic Support*

The AP model supports BSP algorithms that perform accumulative updates, such as WCC (Section 5.2.2), where a vertex does not need all messages from all neighbours to perform its computation (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 SKETCH. (See Appendix B 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. 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 Section 2.4.1, 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 7 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 (Section 2.4.1). 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 7, 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.

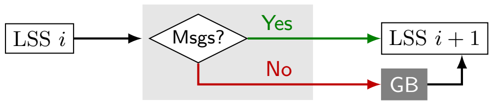
3.1 Local Barriers

For simplicity, we first focus on algorithms with a single computation phase. Algorithms with multiple computation phases are discussed in Section 3.3.

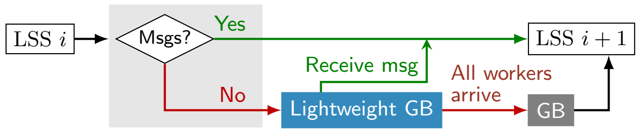
3.1.1 Naive Approach

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 8a). 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



(a) Naive approach.



(b) Improved approach.

Figure 8: Simplified comparison of worker control flows for the two approaches to local barriers. LSS stands for logical superstep and GB for global barrier.

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 (Section 4.3). This prevents the master from erroneously terminating the algorithm.

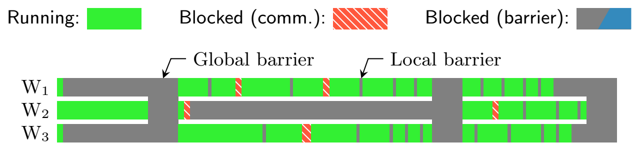
3.1.2 Improved Approach

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 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.

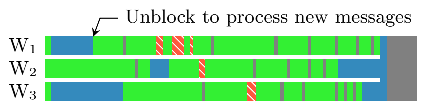
For example, the single source shortest path (SSSP) algorithm begins with only one active vertex (the source), so the number of messages sent between vertices increases, peaks and then decreases with time (Section 5.2.1). Figure 9a shows an SSSP execution using the naive approach, where W_1 and W_3 initially 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 9b). 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 8b). This additional global barrier is lightweight because it is cheaper to block and unblock from compared to the (BSP) global barrier (Section 4.3). Unblocking under the above condition is also efficient because messages arrive in batches (Section 4.2), 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



(a) Naive approach: workers remain blocked.



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

Figure 9: Performance comparison between the naive vs. improved approach, based on SSSP executions of 16 workers on `TW` (Table 1).

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 (Section 5.3). Finally, unlike GraphLab and its GAS model, BAP fully supports graph mutations by having workers resolve pending mutations during a local barrier (Section 4.4).

3.2 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 B 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.

3.3 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 (Section 5.2.3), 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 (Section 3.1.2).

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 $k+1$. $\square$

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 (Section 4.5.1).

Section 4.5 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.

4. 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 [12]. We first provide background on Giraph, before discussing the modifications done to implement the eAP model and then the BAP model.

4.1 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 [25].

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.

4.2 Giraph Async

We first describe Giraph async, our implementation of the enhanced AP (eAP) model that has support for additional types of algorithms (Section 2.4.2). 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. Section 4.5 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. Section 4.4 describes how mutations are supported in GiraphUC.

4.3 Adding Local Barriers

To implement GiraphUC, we add local barriers to Giraph async. We implement local barriers following the improved approach described in Section 3.1.2.

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 Section 3.1.2, 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.

4.4 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 (Section 5.2.3), and algorithms that perform vertex mutations, such as k-core [28, 3]. Even non-mutation algorithms like SSSP (Section 5.2.1) 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.

4.5 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 (Section 5.2.3).

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 Section 3.3, 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 (Section 4.3) 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.

4.5.1 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 (Section 3.3). 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).

4.6 Aggregators and Combiners

Since Giraph is based on the BSP model, aggregators are blocking by default. That is, aggregator values can be ob-

Table 1: Directed datasets. Parentheses indicate values for the undirected versions used by DMST.

Graph	$\lvert V\rvert$	$\lvert E\rvert$	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

tained 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 (Section 4.1).

4.7 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 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.

5. 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 [20]. While Giraph sync and GraphLab sync capture the performance of synchronous systems (BSP and synchronous GAS, respectively), Giraph async is a performant implementation of the eAP model (Section 4.2) and GraphLab async is a state-of-the-art pull-based asynchronous system (asynchronous 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 (Section 4.2). 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 (Section 6) as they are less performant than Giraph and GraphLab [20, 37].

5.1 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][8, 7, 6]. We store all datasets as regular text files on HDFS and load them into all systems using the default random hash partitioning.

Table 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 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 [5, 9].

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.

5.2 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 SSSP and

[1] `http://www.dis.uniroma1.it/challenge9/download.shtml`
[2] `http://law.di.unimi.it/datasets.php`

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 (Section 4.5). We next describe each algorithm in more detail. Appendix A includes their pseudocode.

5.2.1 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 [14]. 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.

5.2.2 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 [21], which starts with all vertices initially active. 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.

5.2.3 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 [13, 30] and undirected versions of our datasets weighted with distinct random edge weights.

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 [13] 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 Section 4.5, these changes are simple and do not affect the algorithm's logic.

5.2.4 PageRank

PageRank is an algorithm that ranks webpages based on the idea that more important pages receive more links from other pages. Like in [34], we use the accumulative update PageRank [40]. 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.

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.

5.3 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 10), we report the mean and 95% confidence intervals of five runs. For PageRank (Figure 11), 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.

5.3.1 SSSP

GiraphUC outperforms all of the other systems for SSSP on all datasets (Figure 10a). This performance gap is particularly large on `US`, which requires thousands of supersteps to complete due to the graph's large diameter (Table 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 10a, 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

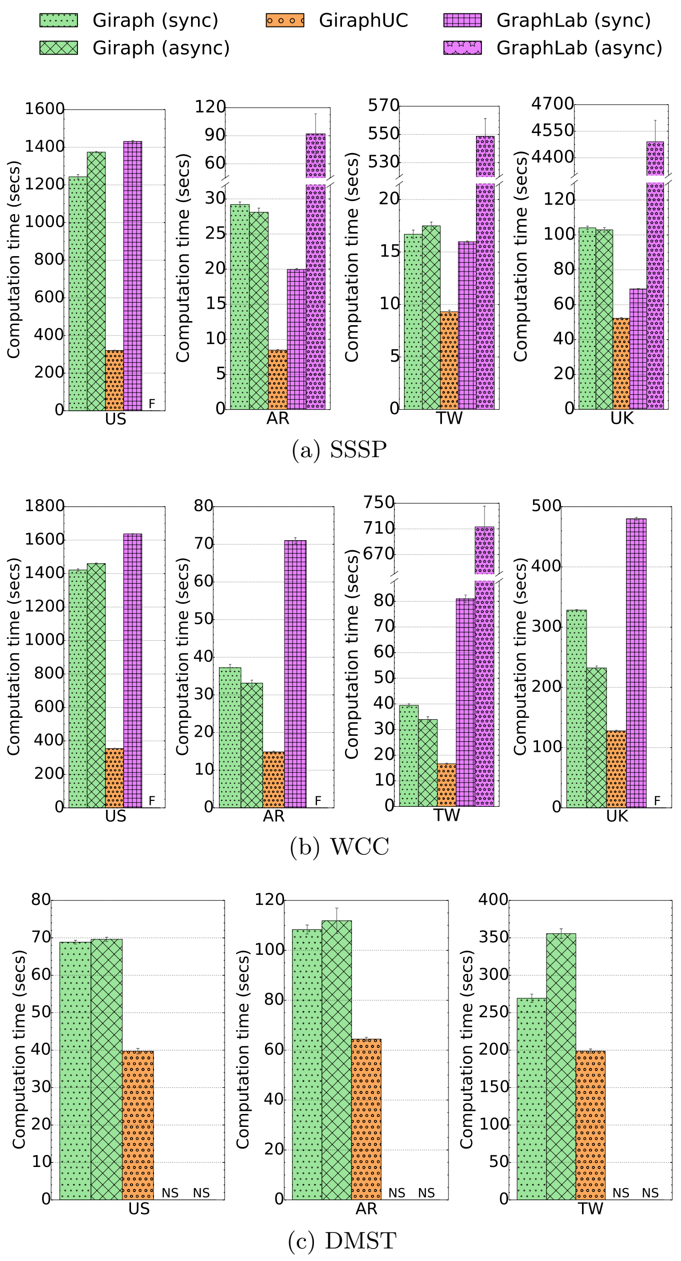


Figure 10: Computation times for SSSP, WCC, and DMST. Missing bars are labelled with 'F' for unsuccessful runs and 'NS' for unsupported algorithms.

highly variable (Figure 10a) 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 (Section 2.3), which results in highly non-deterministic execution compared to GiraphUC's approach of pairing compute threads with partitions (Section 4.1). GraphLab async's poor scalability 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.

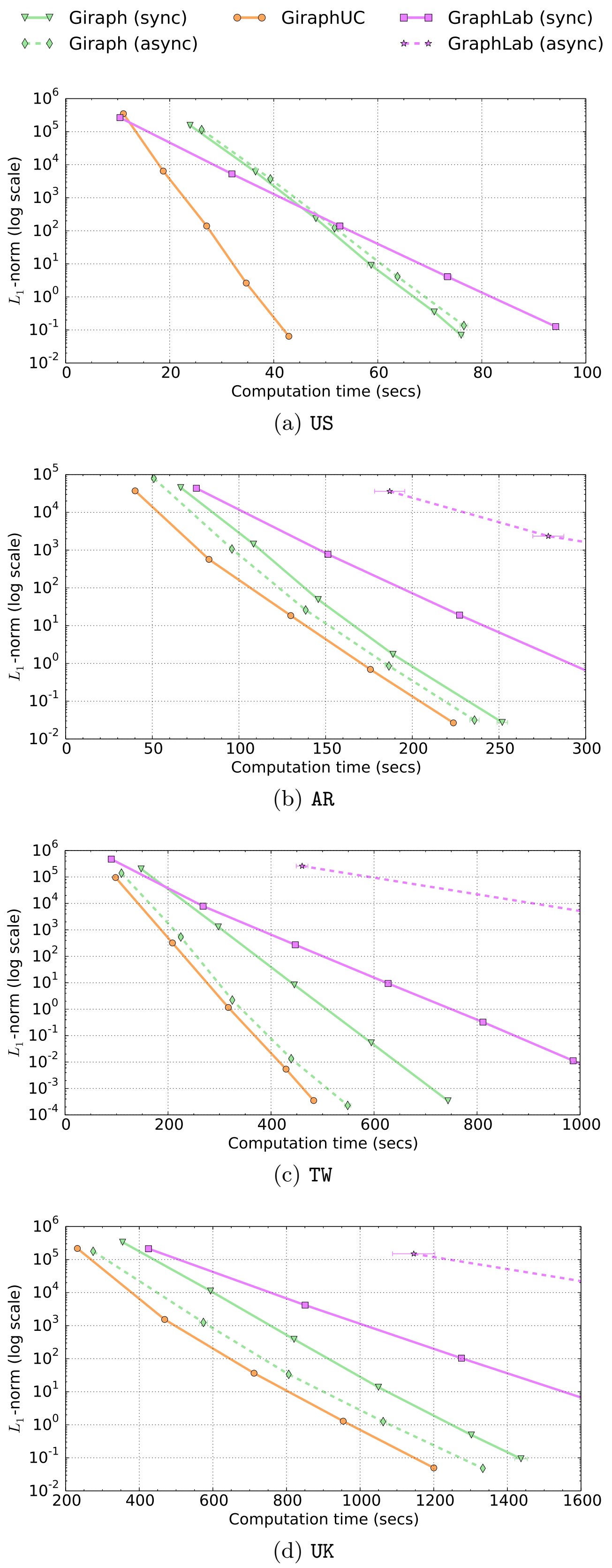


Figure 11: Plots of L_1-norm (error) vs. computation time for PageRank.

5.3.2 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 10b). 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 10b).

5.3.3 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 10c). These performance gains are primarily achieved in the second computation phase of DMST, which typically requires many supersteps to complete (Section 5.2.3). 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 1), meaning DMST will be less compute bound and can hence benefit more under GiraphUC.

Note that GraphLab (both sync and async) cannot implement DMST as they do not fully support graph mutations. This is indicated in Figure 10c with 'NS' for "not supported". Hence, GiraphUC is both performant and more versatile with full support for graph mutations.

5.3.4 PageRank

PageRank, unlike the other algorithms, has a dimension of accuracy in addition to time. Like in [34], 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 [34]. 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 11).

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 11, 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 11a), 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 11b and 11d). 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.

5.4 Sensitivity Analysis

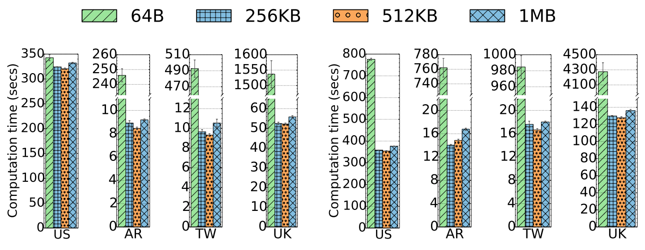
Lastly, we analyze the sensitivity of message batching and the performance of the naive vs. improved approach to local barriers in GiraphUC. All results are again the mean of five runs with 95% confidence intervals.

5.4.1 Message Batching

GiraphUC uses message batching to improve network utilization (Section 4.2). The amount of batching is controlled by the message buffer cache size, which is 512KB by default. Figure 12 shows how varying the buffer cache size from 64 bytes to 256KB, 512KB, and 1MB affects computation time.

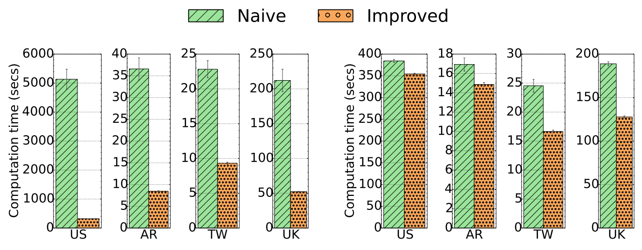
The buffer size of 64 bytes simulates a lack of message batching. This incurs substantial network overheads and long computation times: up to 53× slower than 512KB for SSSP on TW (Figure 12a). For SSSP on US, the performance deterioration is not as pronounced due to SSSP starting with a single active 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 12b). 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



(a) SSSP (b) WCC

Figure 12: Computation times for SSSP and WCC in GiraphUC with different buffer cache sizes.



(a) SSSP (b) WCC

Figure 13: Computation times for SSSP and WCC using naive vs. improved approach to local barriers.

performance. Hence, we stay with the default 512KB buffer cache size for GiraphUC.

5.4.2 Local Barriers

As described in Section 3.1.2, 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 13). 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 workers to unblock, the improved approach suppresses this source of unpredictability and enables superior performance in GiraphUC.

6. RELATED WORK

An overview of Pregel, Giraph, and GraphLab (which incorporates PowerGraph [16]) and how their models compare to GiraphUC's BAP model is provided in Sections 2 and 3.

In addition, there are several other synchronous graph processing systems based on BSP. Apache Hama [2] is a general BSP system that, unlike Giraph, is not optimized for graph processing and does not support graph mutations. Blogel [38] is a BSP graph processing system that extends the vertex-centric programming model by providing a block-centric programming model where, in addition to a vertex compute function, developers can provide a B-compute function that specifies computation on blocks of vertices. GraphX [37] is a system built on the data parallel engine Spark [39] 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's primary goal is to provide more efficient graph processing for end-to-end data analytic pipelines implemented in Spark. Pregelix [11] is a BSP graph processing system implemented in Hyracks [10], a shared-nothing dataflow engine. Pregelix stores graphs and messages as data tuples and uses joins to implement message passing. Hence, it considers graph processing at a lower architectural level. Trinity [32] is a propriety graph computation system that uses a distributed in-memory key-value store to support online graph queries and offline BSP graph processing. GraphChi [23] is a single-machine disk-based graph processing system for processing graphs that do not fit in memory.

There are also several BSP systems that focus on dynamic workload balancing, which is orthogonal to our BAP model. GPS [29] is an open source system that considers optimizations like large adjacency list partitioning and dynamic migration, both of which aim to improve performance on power-law graphs. Unlike Giraph, GPS supports edge mutations but not vertex mutations. GPS was found to be overall less performant than Giraph 1.0.0 and its optimizations did not substantially improve computation times [20]. Our experimental evaluation uses the newer Giraph 1.1.0, which outperforms Giraph 1.0.0. Mizan [22] also considers dynamic migration but its performance was found to be poor compared to Giraph and GraphLab [20]. Catch the Wind [31] is a closed source system built on top of Hama that considers dynamic migration through workload monitoring and message activities.

As outlined in Sections 4.2 and 5, asynchronous execution in BSP has been considered in GRACE and Giraph++. GRACE [36] is a single-machine shared memory system that, unlike Giraph, GiraphUC, and GraphLab, is not distributed. GRACE implements the asynchronous 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, so it does not support graph mutations, and it retains per-superstep global barriers, meaning it does not implement the BAP model. Giraph++ [34], which primarily focuses on a graph-centric programming model, considers a vertex-centric hybrid mode in which local messages are immediately visible but remote messages are delayed until the next superstep. Unlike GiraphUC, Giraph++'s hybrid mode supports neither algorithms that need all messages from all neighbours nor algorithms with multiple computation phases. It also retains global barriers and so does not consider the BAP model.

7. CONCLUSION

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 asynchronous GraphLab and that it is up to 5× faster than synchronous Giraph, asynchronous Giraph, and synchronous GraphLab, and up to 86× faster than asynchronous GraphLab. Thus, GiraphUC enables developers to program their algorithms for the BSP model and transparently execute using the BAP model to maximize performance.

8. REFERENCES

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

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

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

[4] GraphLab: Distributed Graph-Parallel API. http://docs.graphlab.org/classgraphlab_1_1async_ _consistent__engine.html, 2014.

[5] L. Backstrom, P. Boldi, M. Rosa, J. Ugander, and S. Vigna. Four Degrees of Separation. In *WebSci '12*, 2012.

[6] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. UbiCrawler: A Scalable Fully Distributed Web Crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[7] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *WWW '11*, 2011.

[8] P. Boldi and S. Vigna. The Webgraph Framework I: Compression Techniques. In *WWW '04*, 2004.

[9] P. Boldi and S. Vigna. Four Degrees of Separation, Really. http://arxiv.org/abs/1205.5509, 2012.

[10] V. R. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing. In *ICDE '11*, 2011.

[11] Y. Bu, V. Borkar, J. Jia, M. J. Carey, and T. Condie. Pregelix: Big(ger) Graph Analytics on A Dataflow Engine. *PVLDB*, 8(2):161–172, 2015.

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

[13] S. Chung and A. Condon. Parallel Implementation of Borvka's Minimum Spanning Tree Algorithm. In *IPPS '96*.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition.

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

[16] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI '12*, 2012.

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

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

[19] M. Han and K. Daudjee. Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Processing Systems. *PVLDB*, 8(9), 2015.

[20] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin. An Experimental Comparison of Pregel-like Graph Processing Systems. *PVLDB*, 7(12):1047–1058, 2014.

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

[22] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: A system for dynamic load balancing in large-scale graph processing. In *EuroSys '13*, pages 169–182, 2013.

[23] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale Graph Computation on Just a PC. In *OSDI '12*, pages 31–46, 2012.

[24] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB*, 5(8):716–727, 2012.

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

[26] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD/PODS '10*.

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

[28] L. Quick, P. Wilkinson, and D. Hardcastle. Using Pregel-like Large Scale Graph Processing Frameworks for Social Network Analysis. In *ASONAM '12*, 2012.

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

[30] S. Salihoglu and J. Widom. Optimizing Graph Algorithms on Pregel-like Systems. Technical report, Stanford, 2013.

[31] Z. Shang and J. X. Yu. Catch the Wind: Graph Workload Balancing on Cloud. In *ICDE '13*, pages 553–564, 2013.

[32] B. Shao, H. Wang, and Y. Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *SIGMOD '13*, 2013.

[33] S. Suri and S. Vassilvitskii. Counting Triangles and the Curse of the Last Reducer. *WWW '11*, 2011.

[34] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From "Think Like a Vertex" to "Think Like a Graph". *PVLDB*, 7(3):193–204, 2013.

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

[36] G. Wang, W. Xie, A. J. Demers, and J. Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR '13*, 2013.

[37] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. GraphX: A Resilient Distributed Graph System on Spark. In *GRADES '13*, pages 2:1–2:6, 2013.

[38] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A Block-Centric Framework for Distributed Computation on Real-World Graphs. *PVLDB*, 7(14):1981–1992, 2014.

[39] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *HotCloud '10*, 2010.

[40] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Accelerate Large-scale Iterative Computation Through Asynchronous Accumulative Updates. In *ScienceCloud '12*, 2012.

APPENDIX

A. ALGORITHM PSEUDOCODE

Here, we provide the pseudocode for SSSP (Algorithm 1), WCC (Algorithm 2), and PageRank (Algorithm 3). We omit the pseudocode for DMST due to its complexity.

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()
```

Algorithm 2 WCC pseudocode.

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

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()
```

B. PROOFS

In this section, we provide proofs for Theorems 1 and 2.

B.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[3] 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.

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

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:

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 [40] 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 [40], 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 $m = f(v_{\text{value}})$.

B.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.

B.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.

B.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 B.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[4] 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 (Section 4.2). 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 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. □

B.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 B.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,

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

$j \in [0, N)$, 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. $\square$