Providing Serializability for Pregel-like Graph Processing Systems

University of Waterloo Technical Report CS-2016-01*

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

There is considerable interest in the design and development of distributed systems that can execute algorithms to process large graphs. Serializability guarantees that parallel executions of a graph algorithm produce the same results as some serial execution of that algorithm. Serializability is required by many graph algorithms for accuracy, correctness, or termination but existing graph processing systems either do not provide serializability or cannot provide it efficiently. To address this deficiency, we provide a complete solution that can be implemented on top of existing graph processing systems. Our solution formalizes the notion of serializability and the conditions under which it can be provided for graph processing systems. We propose a novel partition-based synchronization approach that enforces these conditions to efficiently provide serializability. We implement our partition-based technique into the open source graph processing system Giraph and demonstrate that our technique is configurable, transparent to algorithm developers, and provides large across-the-board performance gains of up to 26× over existing techniques.

1. INTRODUCTION

Graph data processing has become ubiquitous due to the large quantities of data collected and processed to solve real-world problems. For example, Facebook processes massive social graphs to compute popularity and personalized rankings, find communities, and propagate advertisements for over 1 billion monthly active users [16]. Google processes web graphs containing over 60 trillion indexed webpages to determine influential vertices [19].

Graph processing solves real-world problems through algorithms that are implemented and executed on *graph processing systems*. These systems provide programming and computation models for graph algorithms as well as correctness guarantees that algorithms require.

One key correctness guarantee is *serializability*. Informally, a graph processing system provides serializability if it can guarantee that parallel executions of an algorithm, implemented with its programming and computation models, produce the same results as some serial execution of that algorithm [18].

Serializability is required by many algorithms, for example in machine learning, to provide both theoretical and empirical guarantees for convergence or termination. Parallel algorithms for combinatorial optimization problems experience a drop in performance and accuracy when parallelism is increased without consideration for serializability. For example, the Shotgun algorithm for L_1-regularized loss minimization parallelizes sequential coordinate descent to handle problems with high dimensionality or large sample sizes [11]. As the number of parallel updates is increased, convergence is achieved in fewer iterations. However, after a sufficient degree of parallelism, divergence occurs and *more* iterations are required to reach convergence [11]. Similarly, for energy minimization on NK energy functions (which model a system of discrete spins), local search techniques experience an abrupt degradation in the solution quality as the number of parallel updates is increased [32]. Some algorithms also require serializability to prevent unstable accuracy [27] while others require it for statistical correctness [17]. Graph coloring requires serializability to terminate on dense graphs [18] and, even for sparse graphs, will use significantly fewer colors and complete in only a single iteration (rather than many iterations) when executed serializably.

Providing serializability in a graph processing system is fundamentally a system-level problem that informally requires: (1) vertices see up-to-date data from their neighbors and (2) no two neighboring vertices execute concurrently. The general approach is to pair an existing system or computation model with a *synchronization technique* that enforces conditions (1) and (2). Despite this, of the graph processing systems that have appeared over the past few years, few provide serializability as a configurable option. For example, popular systems like Pregel [28], Giraph [1], and GPS [31] pair a vertex-centric programming model with the bulk synchronous parallel (BSP) computation model [34] but do not provide serializability.

Giraphx [33] provides serializability by pairing the asynchronous parallel (AP) model, which is an asynchronous extension of the BSP model, with the single-layer token passing and vertex-based distributed locking synchronization techniques. However, it implements these synchronization techniques as part of specific user algorithms rather than within the system, meaning algorithm developers must re-implement the techniques into every algorithm that they

*This technical report is the extended version of a paper published in EDBT 2016 [21].

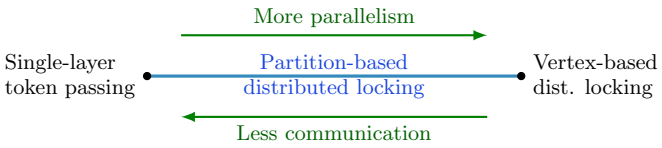


Figure 1: Spectrum of synchronization techniques.

write. Consequently, Giraphx unnecessarily couples and exposes internal system details to user algorithms, meaning serializability is neither a configurable option nor transparent to the algorithm developer. Furthermore, its implementation of vertex-based distributed locking unnecessarily divides each superstep, an iteration of computation, into multiple sub-supersteps in which only a subset of vertices can execute. This exacerbates the already expensive communication and synchronization overheads associated with the global synchronization barriers that occur at the end of each superstep [20], resulting in poor performance.

GraphLab [27], which now subsumes PowerGraph [18], takes a different approach by starting with an asynchronous implementation of the Gather, Apply, Scatter (GAS) computation model. This asynchronous mode (GraphLab async) avoids global barriers by using distributed locking. GraphLab async provides the option to execute with or without serializability and uses vertex-based distributed locking as its synchronization technique. However, GraphLab async suffers from high communication overheads [22, 20] and scales poorly with this technique. Moreover, neither GraphLab nor Giraphx provide a theoretical framework for proving the correctness of their synchronization techniques.

Irrespective of the specific system, synchronization techniques used to enforce conditions (1) and (2) fall on a spectrum that trades off parallelism with communication overheads (Figure 1). In particular, single-layer token passing and vertex-based distributed locking fall on the extremes of this spectrum: token passing uses minimal communication but unnecessarily restricts parallelism, forcing only one machine to execute at a time, while vertex-based distributed locking uses a dining philosopher algorithm to maximize parallelism but incurs substantial communication overheads due to every vertex needing to synchronize with their neighbors.

To overcome these issues, we first formalize the notion of serializability in graph processing systems and establish the conditions under which it can be provided. To the best of our knowledge, no existing work has presented such a formalization. To address the shortcomings of the existing techniques, we introduce a fundamental design shift towards *partition aware* synchronization techniques, which exploit graph partitions to improve performance. In particular, we propose a novel *partition-based distributed locking* solution that allows control over the coarseness of locking and the resulting trade-off between parallelism and communication overheads (Figure 1). We implement all techniques at the system level in the open source graph processing system Giraph so that they are performant, configurable, and transparent to algorithm developers. We demonstrate through experimental evaluation that our partition-based solution substantially outperforms existing techniques.

Our **contributions** are hence threefold: **(i)** we formalize the notion of serializability in graph processing systems and establish the conditions that guarantee it; **(ii)** we introduce the notion of partition aware techniques and our novel partition-based distributed locking technique that enables control over the trade-off between parallelism and communication overheads; and **(iii)** we implement and experimentally compare the techniques with Giraph and GraphLab to show that our partition-based technique provides substantial across-the-board performance gains of up to 26× over existing synchronization techniques.

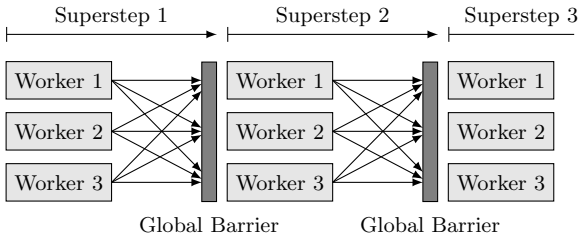


Figure 2: The BSP model, illustrated with three supersteps and three workers [25].

This paper is organized as follows. In Section 2, we provide background on the BSP, AP, and GAS models. In Section 3, we formalize serializability and, in Sections 4 and 5, describe both existing techniques and our partition-based approach. In Section 6, we detail their implementations in Giraph. We present an extensive experimental evaluation of these techniques in Section 7 and describe related work in Section 8 before concluding in Section 9.

2. BACKGROUND AND MOTIVATION

In this section, we introduce the computation models and give a concrete motivation for serializability.

2.1 BSP Model

Bulk synchronous parallel (BSP) [34] is a computation model in which computations are divided into a series of (BSP) *supersteps* separated by global barriers (Figure 2). Pregel (and Giraph) pairs BSP with a vertex-centric programming model, where vertices are the units of computation and edges act as communication channels.

Graph computations are specified by a user-defined compute function that executes, in parallel, on all vertices in each superstep. The function specifies how each vertex processes its received messages, updates its vertex value, and who to send messages to. Importantly, messages sent in one superstep can be consumed/processed by their recipients only in the next superstep. Vertices can vote to halt to become inactive but are reactivated by incoming messages. The computation terminates when all vertices are inactive and no more messages are in transit.

Pregel and Giraph use a master/workers configuration. The master machine partitions the input graph across worker machines, coordinates all global barriers, and performs termination checks based on the two aforementioned conditions. The graph is partitioned by *edge-cuts*: each vertex belongs to a single worker while an edge can span two workers. Finally, BSP is *push-based*: messages are pushed by the sender and buffered at the receiver.

As a running example, consider the greedy graph coloring algorithm. Each vertex starts with the same color (denoted by its vertex value) and, in each superstep, selects the smallest non-conflicting color based on its received mes-

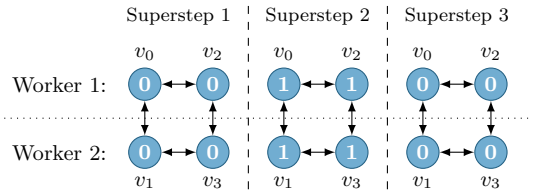


Figure 3: BSP execution of greedy graph coloring. Each graph is the state at the end of that superstep.

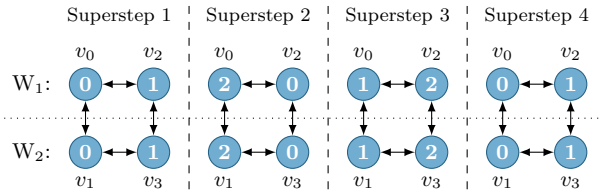


Figure 4: AP execution of greedy graph coloring. Each graph is the state at the end of that superstep.

sages, broadcasts this change to its neighbors, and votes to halt. The algorithm terminates when there are no more color conflicts. Consider an undirected graph of four vertices partitioned across two worker machines (Figure 3). All vertices broadcast the initial color 0 in superstep 1 but the messages are not visible until superstep 2. Consequently, in superstep 2, all vertices update their colors to 1 based on stale data. Similarly for superstep 3. Hence, vertices collectively oscillate between 0 and 1 and the algorithm never terminates. However, if we could ensure that only v_0 and v_3 execute in superstep 2 and only v_2 and v_1 execute in superstep 3, then this problem would be avoided. As we will show in Section 4.3, serializability provides precisely this solution.

2.2 AP Model

The asynchronous parallel (AP) model improves on the BSP model by reducing staleness: instead of delaying all messages until the next superstep, vertices can immediately process any received messages (including ones sent in the same superstep). The AP model retains global barriers to separate supersteps, so messages that arrive too late to be seen by a vertex in superstep i (because the vertex was already executed) will be processed in the next superstep $i+1$. We use a more efficient and performant version of the AP model, described in [20], and its implementation in Giraph, which we will refer to as Giraph async.

Like BSP, the AP model can also fail to terminate for the greedy graph coloring algorithm. Consider again the undirected graph (Figure 4) and suppose that workers W_1 and W_2 execute their vertices sequentially as v_0 then v_2 and v_1 then v_3, respectively. Furthermore, suppose the pairs v_0, v_1 and v_2, v_3 are each executed in parallel. Then the algorithm fails to terminate. Specifically, in superstep 1, v_0 and v_1 initialize their colors to 0 and broadcast to their neighbors. Due to the asynchronous nature of AP, v_2 and v_3 are able to see this message 0 and select the color 1. Similarly, in superstep 2, v_0 and v_1 now see each other's message 0 (sent in superstep 1) and also the message 1 from v_2 and v_3, respectively, so they update their colors to 2. Similarly for v_2 and v_3, who now update their colors to 0. Ultimately, the graph's state at superstep 4 returns to that at superstep 1, so the vertices are collectively cycling through three graph states in an infinite loop.

However, if we can force v_0 to execute concurrently with v_3 instead of v_1 (and v_2 with v_1), then neighboring vertices will not simultaneously pick the same color. Furthermore, if we ensure that v_2 and v_1 wait for the messages from v_3 and v_0 to arrive before they execute, then they will have up-to-date information on all their neighbors' colors. With these two constraints, graph coloring will terminate in just two supersteps. In Section 3, we present a theoretical framework for serializability that formalizes and incorporates these constraints as correctness criteria.

2.3 GAS Model

The Gather, Apply, and Scatter (GAS) model is used by GraphLab for both its synchronous and asynchronous modes, which we refer to as GraphLab sync and GraphLab async. These two system modes use the sync GAS and async GAS models, respectively. In GAS, each vertex pulls information from its neighbors in the gather phase, applies the accumulated data in the apply phase, and updates and activates neighboring vertices in the scatter phase.

Like Pregel and Giraph, GraphLab pairs GAS with a vertex-centric programming model. However, as evidenced by the Gather phase, GAS is *pull-based* rather than push-based. Furthermore, GraphLab partitions graphs by *vertex-cut*: for each vertex u, one worker owns the *primary copy* of u while all other workers owning a neighbor of u get a local read-only replica of u.

Sync GAS is similar to BSP: vertices are executed in supersteps separated by global barriers and the effects of apply and scatter of one superstep are visible only to the gather of the next superstep. Async GAS, however, is different from AP as it has no notion of supersteps. To execute a vertex u, each GAS phase individually acquires a write lock on u and read locks on u's neighbors to prevent data races [3]. However, this does not provide serializability because GAS phases of different vertex computations can interleave [18]. To provide serializability, a synchronization technique must be added on top of async GAS. This technique prevents neighboring computations from interleaving by performing distributed locking over all three GAS phases.

Async GAS can similarly fail to terminate for graph coloring [18]. For example, for the graph in Figure 4, suppose both W_1 and W_2 each have two threads for their two vertices and that all four threads execute in parallel. Then, as described above, the GAS phases of different vertices will interleave, which causes vertices to see stale colors and so the execution is not guaranteed to terminate: it can become stuck in an infinite loop. In contrast, executing in async GAS with serializability will always terminate successfully.

3. SERIALIZABILITY

In this section, we present a theoretical framework that formalizes key conditions under which serializability can be provided for graph processing systems. Later, we show how serializability can be enforced efficiently in these systems.

3.1 Preliminaries

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

systems, irrespective of the computation models they use.

Existing work [18, 33] considers serializability for vertex-centric algorithms where vertices communicate only with their direct neighbors, which is the behaviour of the majority of algorithms that require serializability. For example, the GAS model supports only algorithms where vertices communicate with their direct neighbors [27, 18]. Thus, we focus on this type of vertex-centric algorithms. Our goal is to provide serializability transparently within the graph processing system, independent of the particular algorithm being executed.

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

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

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

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

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

An execution is *serializable* if it produces the same result as a serial execution in which all reads are fresh. Formally, this is one-copy serializability (1SR) [5]. Informally, we will say a system provides serializability if all executions conform to 1SR. In terms of traditional transaction terminology, we define a site as a worker machine, an item as a vertex, and a transaction as the execution of a single vertex. We detail such transactions next.

3.2 Transactions

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

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

Our definition relies only on the fact that the system is vertex-centric and not on the nuances of specific computation models. For example, although BSP and AP have a notion of supersteps, the i for a transaction $T_i(N_u)$ has no relation to the superstep count. The execution of u in two different supersteps is represented by two different transactions $T_i(N_u)$ and $T_j(N_u)$. Our definitions also work when there is no notion of supersteps, such as in async GAS, or when there are per-worker logical supersteps (supersteps that are not globally coordinated), such as proposed in [20]. Thus, the notion of a transaction that follows from our above definition is consistent with the standard notion of a transaction [5]: it captures, for graph processing, the atomic unit of operation that acts on shared data (the graph state).

3.3 Our Approach

In contrast to traditional database systems, graph processing systems present unique constraints that need to be taken into account for providing serializability.

First, Pregel-like graph processing systems such as Giraph and GraphLab do not natively support transactions: they are not database systems and thus have no notion of commits or aborts. The naive solution is to implement transaction support into all graph processing systems. However, this requires a fundamental redesign of each system, which is neither general nor reusable. Moreover, such a solution fails to be modular: it introduces performance penalties for graph algorithms that do not require serializability.

Second, an abort in a graph processing system can result in prohibitively expensive (and possibly cascading) rollbacks on the distributed graph state: a transaction often involves sending messages to vertices of different worker machines, the effects of which are difficult to undo. Consequently, solutions relying on optimistic currency control are a poor fit for graph processing due to the high cost of aborts.

However, for graph processing systems, a *write-all* approach [5] can be used to keep replicas up-to-date because graph processing systems replicate only for distributed computation and not for availability. When a worker machine fails, we lose a portion of the input graph and so cannot proceed with the computation. Indeed, failure recovery requires all machines to rollback to a previous checkpoint [1, 27, 28], meaning the problem of pending writes to failed machines never occurs. In contrast, a write-all approach is very expensive for traditional database systems because they replicate primarily for better performance and/or availability.

Furthermore, as detailed in Section 3.2, the read and write sets of each transaction are known a priori (N_v and v, respectively, for a transaction $T_i(N_v)$), which means pessimistic concurrency control can be used to avoid costly aborts.

Our approach, at a high level, is to pair graph processing

systems with a *synchronization technique*, which uses (1) a write-all approach to avoid data staleness and (2) pessimistic concurrency control to prevent conflicting transactions from starting. For the graph processing systems, (1) means vertices will always read from fresh replicas and so the system need not reason about versioning, while (2) means all transactions that start will commit, so aborts never occur and hence the system can treat all operations as final without needing explicit support for commits and aborts. Furthermore, this solution enables us to use transactions to formally reason about correctness without the burden of fundamentally redesigning each system to support transactions. Since aborts cannot occur, we also avoid the expensive penalties of distributed cascading rollbacks.

Using the definitions introduced in Section 3.2, we can formalize our requirements into the following two conditions:

CONDITION C1. *Before any transaction $T_i(N_u)$ executes, all replicas $v \in N_u$ are up-to-date.*

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

Next, we will prove that 1SR can be provided by enforcing these two conditions.

3.4 Correctness

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

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

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

We will use the standard notion of conflicts and histories [5]. Two transactions conflict if they have conflicting operations. A *serial single-copy history* produced by the serial execution of an algorithm is a sequence of transactions where operations act on a single logical copy and do not interleave. For example, one possible serial single-copy history for two vertices u and v would be

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

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

We can then prove a useful relationship between input and serializability graphs (Lemma 2) before formally defining the relationship between serializability and conditions C1 and C2 (Theorem 1).

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

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

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

Recall that the serialization graph of a history consists of a node for each transaction and an edge from T_i to T_j if one of T_i's operations precedes and conflicts with one of T_j's operations [5]. Since all reads occur before writes, every transaction T_i's read operation $r_i[N_{u_i}]$ will precede and conflict with transaction T_{i-1}'s write operation $w_{i-1}[u_{i-1}]$ for $i \in (1, n]$ because $u_{i-1} \in N_{u_i}$. That is, there exists a directed path from T_n to T_1 in the serialization graph. Additionally, because of the directed cycle in the input graph, we also have a conflict due to $u_n \in N_{u_1}$. This adds an edge from T_1 to T_n which, together with the directed path from T_n to T_1, creates a directed cycle in the serialization graph. Hence, all histories of this form produce serialization graphs with a directed cycle.[1]

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

The former ordering can only be due to $r_j[N_v] < w_i[u]$, since every transaction has its read precede its write, which implies that $u \in N_v$ and so there exists an edge (u, v) in the input graph. Similarly, the latter ordering implies that there is a directed path from v to u. For example, suppose the ordering is $T_i(N_u) < T_k(N_w) < T_j(N_v)$. Then this must arise due to $r_i[N_u] < w_k[w]$ and $r_k[N_w] < w_j[v]$, meaning $w \in N_u$ and $v \in N_w$. Equivalently, there exist edges (w, u) and (v, w) in the input graph and thus a directed path from v to u. In general, the more transactions there are between $T_i(N_u)$ and $T_j(N_v)$, the longer this path.

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

THEOREM 1. *All executions are serializable for all input graphs if and only if conditions C1 and C2 are both true.*

PROOF. (IF) Since condition C1 is true, by Lemma 1 we can apply standard serializability theory. By the serializability theorem [5], it suffices to show if condition C2 is true then all possible histories have an acyclic serialization graph.

Consider two arbitrary concurrent transactions $T_i(N_u)$ and $T_j(N_v)$. For vertex-centric systems, each vertex is always executed by exactly one thread of execution because

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

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

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

the compute function is written for serial execution. That is, we must have $u \neq v$ for T_i and T_j. Since condition C2 is true, the write operations of T_i and T_j are always on $u \notin N_v$ and $v \notin N_u$, respectively, meaning the transactions do not conflict. That is, condition C2 eliminates all conflicting transactions from all possible histories, so the serialization graph must be acyclic. Since we make no assumptions about the input graph, this holds for all input graphs.

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

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

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

Lastly, we note a theoretically interesting corollary.

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

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

However, in practice, Corollary 1 is insufficient for providing serializability because nearly all real world graphs have at least one undirected edge. Moreover, the synchronization technique used to enforce condition C1 will also enforce condition C2 (Sections 3.3 and 3.5).

3.5 Enforcing Serializability

The computation models from Section 2 do not enforce conditions C1 and C2 and therefore, by Theorem 1, do not provide serializability. Consequently, graph processing systems that implement these models also do not provide serializability. Moreover, these models do not guarantee fresh reads even under serial executions (on a single machine or under the sequential execution of multiple machines). For example, BSP effectively updates replicas lazily[3] because messages sent in one superstep, even if received, cannot be read by the recipient in the same superstep. Thus, both m-boundary *and* m-internal vertices (Definition 1) suffer stale reads under a serial execution. While AP reduces this staleness and can update local replicas eagerly, it propagates messages to remote replicas lazily without the guarantees of condition C1 and so stale reads can again occur under a serial execution of multiple machines.

As mentioned in Section 3.3, to provide serializability, we enforce conditions C1 and C2 by adding a synchronization technique (Sections 4 and 5) to the systems that implement the above computation models. These synchronization techniques implement a write-all approach for updating replicas, which is required for enforcing condition C1. They also ensure that a vertex u does not execute concurrently with any of its in-edge *and* out-edge neighbors. At first glance, this appears to be stronger than what condition C2 requires. However, suppose v is an out-edge neighbor of u and v is currently executing. Then if u does not synchronize with its out-edge neighbors, it will erroneously execute concurrently with v, violating condition C2 for v. Alternatively, if v is an out-edge neighbor of u then u is an in-edge neighbor of v, so they must not execute concurrently.

4. EXISTING SYNCHRONIZATION TECHNIQUES

Token passing and distributed locking are the two general approaches for implementing synchronization techniques that enforce conditions C1 and C2. In this section, we review two existing synchronization techniques: single-layer token passing and vertex-based distributed locking.

4.1 Preliminaries

How a synchronization technique implements a write-all approach (Section 3.3) depends on whether the computation model is synchronous or asynchronous.

In asynchronous computation models (AP and async GAS), replicas immediately apply received updates. Thus, local replicas can be updated eagerly, since there is no network communication (Section 6). Remote replicas, however, are updated lazily in a just-in-time fashion to provide communication batching. This lazy update is possible because all vertices are coordinated by a synchronization technique: any vertex v must first acquire a shared resource (e.g., a token or a fork) from its neighbor u before it can execute. Consequently, for an m-boundary vertex u with a replica on its neighbor v's worker, u's worker can buffer remote replica updates until v wants to execute (i.e., requests the shared resource)—at which point u's machine will flush all pending remote replica updates (and ensure their receipt) before handing over the shared resource that allows v to proceed.

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

[2]Self-cycles in the input graph are permitted because a single vertex is executed by one thread of execution at any time (see the proof of Theorem 1).

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

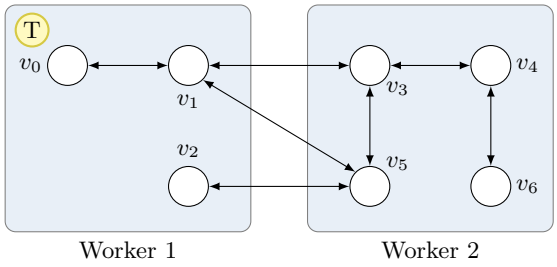


Figure 5: Single-layer token passing, with the global token T at worker 1.

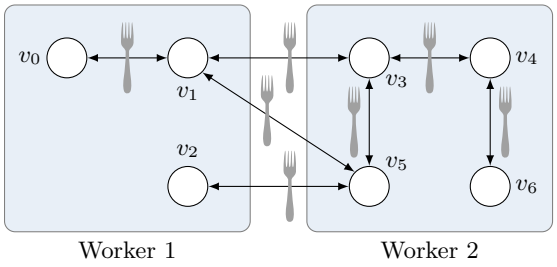


Figure 6: Vertex-based distributed locking.

synchronization techniques for systems with asynchronous computation models, as detailed further in Section 6.

4.2 Single-Layer Token Passing

Single-layer token passing, considered in [33], is a simple technique that passes an exclusive global token in a round-robin fashion between workers arranged in a logical ring. Each worker machine must execute with only one thread.

The worker machine holding the global token can execute both its m-internal and m-boundary vertices (Definition 1), while workers without the token can execute only their m-internal vertices. This prevents neighboring vertices from executing concurrently since each m-internal vertex and its neighbors are executed by a single thread, so there is no parallelism, while an m-boundary vertex can execute only when its worker machine holds the exclusive token. For example, in Figure 5, worker 1 holds the token and so can execute all its vertices, while worker 2 can execute only v_4 and v_6.

To enforce condition C1, local replicas must be updated eagerly, while remote replicas of a worker's m-boundary vertices can be updated in batch before a worker passes along the global token (as updates will arrive before the token). Per Section 4.1, this is possible with asynchronous computation models. Thus, for asynchronous models, single-layer token enforces conditions C1 and C2 and, by Theorem 1, provides serializability. However, this technique does not provide serializability for synchronous computation models as they cannot update local replicas eagerly.

4.3 Vertex-based Distributed Locking

Vertex-based distributed locking, unlike token passing, pairs threads with individual vertices to allow all vertices to attempt to execute in parallel. As motivated in Section 2.1, the key idea is to coordinate these vertices such that neighboring vertices do not execute concurrently, while also addressing issues such as deadlock and fairness.

This coordination is achieved using the Chandy-Misra algorithm [13], which solves the hygienic dining philosophers problem, a generalization of the dining philosophers problem. In this problem, each philosopher is either thinking, hungry, or eating and must acquire a shared fork from each of its neighbors to eat. Philosophers can communicate with their neighbors by exchanging forks and request tokens for forks. The "dining table" is effectively an undirected graph where each vertex is a philosopher and each edge is associated with a shared fork (Figure 6): a philosopher u must acquire $\deg(u)$ forks to eat. The Chandy-Misra algorithm ensures no neighbors eat at the same time, guarantees fairness (no philosopher can hog its forks), and prevents deadlocks and starvation [13]. Hence, condition C2 is enforced.

To enforce condition C1, local replicas are updated eagerly and, for remote replicas, each worker flushes its pending remote replica updates before any m-boundary vertex relinquishes a fork to a vertex of another worker. Then, per Section 4.1, vertex-based distributed locking provides serializability for asynchronous computation models. As we mentioned in Section 4.1, this solution is incompatible with synchronous models (BSP and sync GAS) because these models do not allow local replicas to be updated eagerly. However, applying the theory developed in Section 3, Proposition 1 shows that a constrained vertex-based locking solution can provide serializability for systems with synchronous models..

PROPOSITION 1. *Vertex-based distributed locking enforces conditions C1 and C2 for synchronous computation models when the following two properties hold: (i) all vertices act as philosophers and (ii) fork and token exchanges occur only during global barriers.*

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

5. PARTITION AWARE TECHNIQUES

In this section, we show how partition aware synchronization techniques can address severe limitations of existing techniques. We then present our partition-based solution to demonstrate its significant performance advantages.

5.1 Preliminaries

Existing graph processing systems provide parallelism at each worker machine by pairing computation threads with either graph partitions or individual vertices.

For example, both Giraph and Giraph async (Section 2.2) assign multiple graph partitions to each worker machine and pair threads, each roughly equivalent to a CPU core, with

available partitions. This allows multiple partitions to execute in parallel, while vertices in each partition are executed sequentially. We call such systems *partition aware*.

In contrast, GraphLab async uses over-threading to pair lightweight threads (called fibers) with individual vertices. Thus, it has no notion of partitions. The large number of fibers provides a high degree of parallelism and ensures that CPU cores are kept busy even when some fibers are blocked on communication.

Systems like GraphLab async are well-suited for very fine-grained synchronization techniques such as vertex-based distributed locking (Section 4.3). Partition aware systems like Giraph async are able to support partition aware techniques that, as we will show, take advantage of partitions to significantly improve performance. Since GraphLab async is not partition aware, it is unable to support such techniques.

Lastly, as we will show in the following sections, it is important for synchronization techniques implemented in partition aware systems to distinguish between p-internal and p-boundary vertices, defined as follows.

DEFINITION 4. *A vertex u is a* partition boundary *vertex, or* p-boundary *for short, if at least one of its neighbors v belongs to a different partition from u. Otherwise, u is a* partition internal*, or* p-internal*, vertex.*

5.2 Motivation

The two existing synchronization techniques described in Section 4 suffer from several major performance issues.

Token passing has minimal communication overheads but very limited parallelism (Figure 1): only one worker machine can execute its m-boundary vertices at any time. Having only one global token also results in poor scalability, because the size of the token ring increases with the number of workers, which leads to longer wait times. Moreover, the token ring is fixed: workers that are finished must still receive and pass along the token, which adds unnecessary overheads. This is especially evident in algorithms such as SSSP, where workers may dynamically halt or become active depending on the state of their constituent vertices. Thus, as we show in Section 7.3, single-layer token passing is too coarse-grained, which negatively impacts performance.

On the other hand, vertex-based distributed locking maximizes parallelism, by allowing all vertices to execute in parallel, but suffers significant communication overheads. Vertex-based locking requires, in the worst case, $O(|\mathcal{E}|)$ forks, where $|\mathcal{E}|$ is the number of edges in the graph ignoring directions (i.e., counting undirected edges once). This leads to significant communication overheads due to the forks and corresponding request tokens that must be sent between individual vertices. Furthermore, it is difficult to form large batches of messages (remote replica updates) as messages must be flushed very frequently, whenever an m-boundary vertex releases its forks. Although systems such as GraphLab async can use fibers to try to mask communication latency with additional vertex computations, it does not fully mitigate the communication overheads, which results in poor performance and scalability as we demonstrate in Section 7.3.

A key deficiency of these techniques is that they are not partition aware: given a partition aware system, they are unable to exploit partitions to improve performance. For example, single-layer token passing would pass the global token between the partitions rather than workers, with p-boundary vertices requiring the token to execute (Definition 4). This increases the size of the token ring and does not solve the existing performance problems. Similarly, vertex-based distributed locking (for asynchronous models) would require only p-boundary vertices to act as philosophers, since p-internal vertices are executed sequentially. However, although this reduces the number of forks, the heavy-weight threads will block an entire CPU whenever a vertex blocks on communication. Consequently, it is unable to mask communication latency and performs worse than GraphLab async's pairing of fibers with individual vertices (Section 5.1).

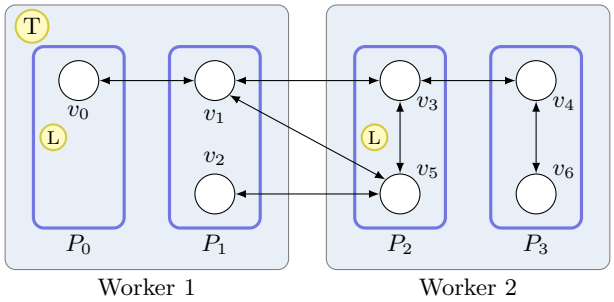


Figure 7: Dual-layer token passing, with the global token T at worker 1 and the local tokens L at partitions 0 and 2.

We address these performance deficiencies by considering *partition aware* synchronization techniques. Adding partition awareness enables us to devise either a more fine-grained token passing technique to increase parallelism, or a more coarse-grained distributed locking technique to reduce communication overheads. We present these approaches next.

5.3 Dual-Layer Token Passing

We propose dual-layer token passing, which, unlike single layer token passing, supports multithreading by being partition aware. This enables more vertices to execute in parallel while ensuring condition C2 is enforced.

Dual-layer token passing uses two layers of tokens and a more fine-grained categorization of vertices. Let u be a vertex of partition P_u of worker W_u. Then an m-internal vertex u is now either a p-internal vertex, if all its neighbors belong to P_u, or a *local boundary* vertex otherwise. An m-boundary vertex u is either *remote boundary*, if its neighbors are only on partitions of other workers, or *mixed boundary* otherwise (i.e., its neighbors belong to partitions of both W_u and other workers). For example, in Figure 7, v_6 is a p-internal vertex, v_0 and v_4 are local boundary vertices, v_2 is a remote boundary vertex, and v_1, v_3, and v_5 are mixed boundary vertices.

A global token is passed in a round-robin fashion between the workers. Each worker also has its own local token passed between its partitions in a round-robin fashion (Figure 7). A p-internal vertex can execute without tokens, while a local boundary vertex requires its partition to hold the local token. A global boundary vertex requires its worker to hold the global token and a mixed boundary vertex requires both tokens to be held. To ensure that every mixed boundary vertex gets a chance to execute, each worker must hold the global token for a number of iterations equal to the number of partitions it owns. Like single-layer token passing, local replicas are updated eagerly while remote replicas are updated before a worker relinquishes the global token. Hence,

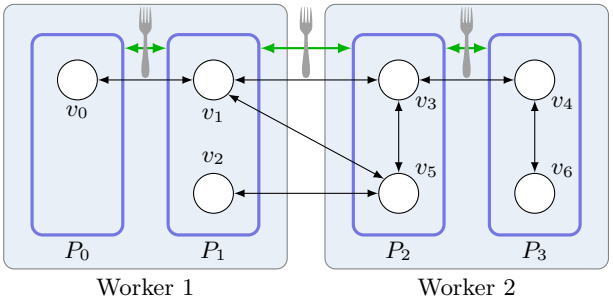


Figure 8: Partition-based distributed locking.

dual-layer token passing enforces conditions C1 and C2 for asynchronous computation models. Then, by Theorem 1, it provides serializability for asynchronous models.

Although dual-layer token passing improves parallelism by adding support for multithreading, it still suffers from the same performance issues as single-layer token passing. It again uses only one global token, has a fixed token ring, and scales poorly when the number of workers and/or partitions are increased. Having only one local token per worker also means local boundary vertices cannot execute in parallel.

While these problems may be solved via more sophisticated schemes, such as using multiple global tokens for more parallelism or tracking additional state to support a dynamic ring, it becomes much harder to guarantee correctness (no deadlocks and no starvation) while also ensuring fairness. Thus, rather than make token passing even more complex and fine-grained, we propose an inherently partition-based, coarse-grained distributed locking approach next.

5.4 Partition-based Distributed Locking

We propose partition-based distributed locking by building on the Chandy-Misra algorithm and treating partitions as the philosophers. Two partitions share a fork if an edge connects their constituent vertices. For example, in Figure 8, partitions P_0 and P_1 share a fork due to the edge between their vertices v_0 and v_1, respectively. Alternatively, forks are associated with the virtual partition edges (in green), created based on the edges between each partition's vertices.

Condition C2 is enforced for p-boundary vertices because neighboring partitions never execute concurrently, while p-internal vertices do not need coordination as each partition is executed sequentially. As an optimization, we can avoid unnecessary fork acquisitions by skipping the partitions for which all vertices are halted and have no more messages. To enforce condition C1, local replicas are updated eagerly and, for remote replicas, each worker flushes its pending remote replica updates before any partition (with an m-boundary vertex) relinquishes a fork to a partition of another worker. Since both conditions are enforced, Proposition 2 follows immediately.

PROPOSITION 2. *Partition-based distributed locking enforces conditions C1 and C2 for asynchronous computation models.*

Hence, by Theorem 1, partition-based distributed locking provides serializability for asynchronous computation models. Synchronous models are not supported as they cannot update local replicas eagerly (Section 4.1), which is required due to the sequential execution of p-internal vertices.

Partition-based distributed locking needs at most $O(|P|^2)$ forks, where $|P|$ is the total number of partitions. By controlling the number of partitions, we can control the granularity of parallelism. On one extreme, $|P| = |V|$ can give vertex-based distributed locking (Section 4.3). On the other extreme, we can have exactly one partition per worker. This latter extreme still provides better parallelism than single-layer token passing because any pair of non-neighboring workers can execute in parallel, with a negligible increase in communication. In general, $|P|$ is set such that each worker can use multithreading to execute multiple partitions in parallel.

Due to this flexibility, partition-based locking is both more general and more performant than vertex-based locking: any choice of $|P| \ll |V|$ significantly reduces the number of forks and hence communication overheads. Moreover, partition-based locking enables messages (remote replica updates) of an entire partition of vertices to be batched, substantially reducing communication overheads. Compared to token passing, partition-based locking enables more parallelism: forks are required only between partitions that cannot execute in parallel, removing the need for a token ring, and halted partitions do not need their forks and will not perform unnecessary communication with their neighbors. These factors result in partition-based locking's superior performance and scalability over both vertex-based distributed locking and token passing.

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

6. IMPLEMENTATION

We now describe our implementations for dual-layer token passing and partition-based distributed locking in Giraph, an open source graph processing system. Each technique is an option that can be enabled and paired with Giraph async to provide serializability. We show in Section 6.5 that providing serializability for AP does not impact usability. We do not consider the constrained vertex-based locking for BSP (Proposition 1) as it further exacerbates BSP's already expensive communication and synchronization overheads [20].

We use Giraph because it is a popular and performant system used, for example, by Facebook [14]. It is partition aware and thus can support partition aware synchronization techniques. We do not implement token passing and partition-based distributed locking in GraphLab async because, as described in Section 5.1, GraphLab async is optimized for vertex-based distributed locking and is *not* partition aware. Adding partitions would require substantial changes to the architecture, design, and functionality of GraphLab async, which is not the focus of this paper.

6.1 Giraph Background

As described in Section 5.1, Giraph assigns multiple graph partitions to each worker. During each superstep, each worker creates a pool of compute threads and pairs available threads with uncomputed partitions. Each worker maintains a message store to hold all incoming messages, while each compute thread uses a message buffer cache to batch outgoing mes-

sages to more efficiently utilize network resources. These buffer caches are automatically flushed when full but can also be flushed manually. In Giraph async, messages between vertices of the same worker skip this cache and go directly to the message store so that they are immediately available for their recipients to process.

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

6.2 Dual-Layer Token Passing

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

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

As per Section 5.3, the global and local tokens are passed in a round-robin fashion. Each local token is passed among its worker's partitions at the end of each superstep. Importantly, each worker holds the global token for n supersteps, where n is the number of partitions owned by that worker. This ensures that every partition's mixed boundary vertices are executed (Section 5.3). Without this, acquiring both tokens becomes a race condition, leading to starvation for some mixed boundary vertices. Since local messages (between vertices of the same worker) are not cached, local replicas are updated eagerly. For remote replicas, workers flush and await delivery confirmations for their remote messages before passing along the global token.

In contrast, Giraphx [33] implements single-layer token passing and as a part of user algorithms rather than within the system. Giraphx stores type information with the vertex objects and uses two supersteps to update vertex types based on their in-edge dependencies. As discussed previously, the former is less performant while the latter wastes two supersteps of computation. Furthermore, this unnecessarily clutters user algorithms with system-level concerns and must be re-implement into every new user algorithm. Thus, it is neither transparent nor configurable.

6.3 Partition-based Distributed Locking

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

In the Chandy-Misra algorithm, forks and tokens must be placed such that the precedence graph, whose edge directions determine which philosopher has priority for each shared fork, is initially acyclic [13]. A simple way to ensure this is to assign each philosopher an id and, for each pair of neighbors, give the token to the philosopher with the smaller id and the dirty fork to the one with the larger id. This guarantees that philosophers with smaller ids initially have precedence over all neighbors with larger ids, because a philosopher must give up a dirty fork upon request (except while it is eating). Partition ids naturally serve as philosopher ids, allowing us to use this initialization strategy.

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

Partitions acquire their forks synchronously by blocking until all forks arrive. This is because even if all forks are available, it takes time for them to arrive over the network, so immediately returning is wasteful and may prevent other partitions from executing (a partition cannot give up clean forks: it must first execute and dirty them). Finally, per Section 5.4, each worker flushes its remote messages before a partition sends a shared fork to another worker's partition.

6.3.1 Vertex-based Distributed Locking

Using the insights from our implementation of partition-based distributed locking, we can also implement vertex-based distributed locking, which is the special case where $|P| = |V|$ (Section 5.4). Each worker tracks fork and token states for its p-boundary vertices and uses vertex ids as the keys for its dual-layer hash map. Keeping this data in a central per-worker data structure, rather than at each vertex object, is even more important than in token passing: forks and tokens are constantly exchanged so their states must be readily available to modify. Storing this data at each vertex object would incur significant deserialization overheads. Fork and token access patterns are also fairly random, which would further incur an expensive traversal of a byte array to locate the desired vertex.

Like the partition-based approach, for directed graphs, each vertex v broadcasts to its out-edge neighbors u so that u can record the in-edge dependency into the per-worker hash map. This occurs during input loading and all messages are batched. Vertices acquire their forks synchronously and each worker flushes its remote messages before any m-boundary vertex forfeits a fork to a vertex of another worker. However, as we show in Section 7, these batches of remote messages are far too small to avoid significant communication overheads.

In contrast, Giraphx's implementation of vertex-based locking sends forks and tokens as part of messages generated by user algorithms rather than as internal system messages. Consequently, forks and tokens sent between different workers are delivered only during global barriers, which unnec-

essarily divides each superstep into multiple sub-supersteps (akin to Proposition 1). The resulting increase in global barriers negatively impacts performance. Our implementation avoids this (Section 6.5). Furthermore, Giraphx again requires the technique to be re-implemented in every new algorithm and is thus neither transparent nor configurable.

6.4 Fault Tolerance

For fault tolerance, we use the existing checkpointing mechanism of Giraph. In addition to the data that Giraph already writes to disk at each synchronous checkpoint, we change Giraph to also record the relevant data structures (hash sets or hash maps) that are used by the synchronization techniques. For dual-layer token passing, each worker also records whether they have the global token and the id of the partition holding the local token. Checkpoints occur after a global barrier and thus capture a consistent state: there are no vertices executing and no in-flight messages. Thus, neither token passing's global token nor distributed locking's fork and request tokens are in transit.

6.5 Algorithmic Compatibility and Usability

A system can provide one computation model for algorithm developers to code with and use a different computation model to execute user algorithms. For example, Giraph async allows algorithm developers to code for the BSP model and transparently execute with an asynchronous computation model to maximize performance [20]. Thus, with respect to BSP, the more efficient AP model implemented by Giraph async does not negatively impact usability.

When we pair Giraph async with partition-based or vertex-based distributed locking, it remains backwards compatible with (i.e., can still execute) algorithms written for the BSP model. To take advantage of serializability, algorithm developers can now code for a serializable computation model. Specifically, this is the AP model with the additional guarantee that conditions C1 and C2 hold. For example, our graph coloring algorithm is written for this serializable AP model rather than for BSP (Section 7.2.1).

However, not all synchronization techniques provide this clean abstraction. Token passing fails in this regard because only a subset of vertices execute in each superstep. That is, token passing cannot provide the guarantee that all vertices will execute some code in superstep i, because only a subset of the vertices will execute at superstep i. The same issue arises for the constrained vertex-based distributed locking solution for BSP and sync GAS (Proposition 1) and Giraphx's implementation of vertex-based distributed locking (Section 6.3.1), because they rely on global barriers for the exchange of forks and tokens. In contrast, our implementations of partition-based and vertex-based locking ensure that all vertices are executed exactly once in each superstep and thus provide superior compatibility and usability.

7. EXPERIMENTAL EVALUATION

We compare dual-layer token passing and partition-based distributed locking using Giraph async and vertex-based distributed locking using GraphLab async. We exclude Giraph async for vertex-based locking because it is much slower than GraphLab async, up to 44× slower on `OR` (Table 1). As discussed in Section 5.1, this is because GraphLab async is specifically tailored for the vertex-based technique whereas Giraph async is not. On the other hand, unlike Giraph async, GraphLab async is not partition aware and thus cannot support token passing or partition-based distributed locking. Hence, our evaluation focuses on the most performant combinations of systems and synchronization techniques.

Table 1: Directed datasets. Parentheses give values for the undirected versions used by graph coloring.

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

7.1 Experimental Setup

To evaluate the different synchronization techniques, we use 16 and 32 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 implement our modifications in Giraph 1.1.0-RC0 and compare against GraphLab 2.2, which is the latest version that provides serializability.

We use large real-world datasets[4,5][8, 7, 6], which are stored on HDFS as text files and loaded into each system using the default random hash partitioning. We use hash partitioning as it is the fastest method of partitioning datasets across workers and, importantly, does not favour any particular synchronization technique. Alternative partitioning algorithms such as METIS [24] are impractical as they can take several hours to partition large datasets [22, 30].

Table 1 lists the four graphs we use: `OR` and `TW` are social network graphs while `AR` and `UK` are web graphs. $|V|$ and $|E|$ of Table 1 denote the number of vertices and directed edges for each graph, while the maximum degree gives a sense of how skewed the graph's degree distribution is. All graphs have large maximum degrees because they follow a power-law degree distribution.

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

7.2 Algorithms

We use graph coloring, PageRank, SSSP, and WCC as our algorithms. Our choice is driven by the requirements exhibited by graph processing algorithms that need serializability. As described in Section 1, many machine learning algorithms require serializability for correctness and convergence. SSSP is a key component in reinforcement learning while WCC is used in structured learning [29, 10]. Both algorithms are used with extensive parallelism, making con-

[4] `http://snap.stanford.edu/data/`

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

vergence a crucial criterion that serializability can provide. Similarly, as established in Section 2, graph coloring falls into yet another class of algorithms where serializability ensures successful termination. Finally, PageRank is a good comparison algorithm for two reasons: first, existing systems that have considered serializability also implement PageRank [33, 18] and second, the simple computation and communication patterns of PageRank are identical to other more complex algorithms [4], which allows us to better understand the performance of the synchronization techniques without being hindered by algorithmic complexity.

7.2.1 Graph Coloring

We use a greedy graph coloring algorithm (Algorithm 1) that requires serializability and an undirected input graph. Each vertex u initializes its value/color as NO_COLOR. Then, based on messages received from its (in-edge) neighbors, u selects the smallest non-conflicting color as its new color and broadcasts it to its (out-edge) neighbors.

Algorithm 1 Graph coloring pseudocode.

```
1 procedure COMPUTE(vertex, incoming messages)
2     if superstep == 0 then
3         vertex.setValue(NO_COLOR)
4         return
5     if vertex.getValue() == NO_COLOR then
6         c_min ← smallest non-conflicting color
7         vertex.setValue(c_min)
8         Send c_min to vertex's out-edge neighbors
9     voteToHalt()
```

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

7.2.2 PageRank

PageRank is an algorithm that ranks webpages based on the idea that more important pages receive more links from other pages. Each vertex u starts with a value of 1.0. At each superstep, u updates its value to $pr(u) = 0.15 + 0.85x$, where x is the sum of values received from u's in-edge neighbors, and sends $pr(u)/\deg^+(u)$ along its out-edges. The algorithm terminates after the PageRank value of every vertex u changes by less than a user-specific threshold between two consecutive execution of u. The output $pr(u)$ gives the expectation value for a vertex u, which can be divided by the number of vertices to obtain the probability value.

We use a threshold of 0.01 for OR and AR and 0.1 for TW and UK so that experiments complete in a reasonable amount of time. Using the same threshold ensures that all systems perform the same amount of work for each graph.

7.2.3 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 [15]. 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 neighbors and propagate any newly discovered minimum distance to all neighbors. We use unit edge weights and the same source vertex to ensure that all systems perform the same amount of work.

7.2.4 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 [23], 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 neighbors.

7.3 Results

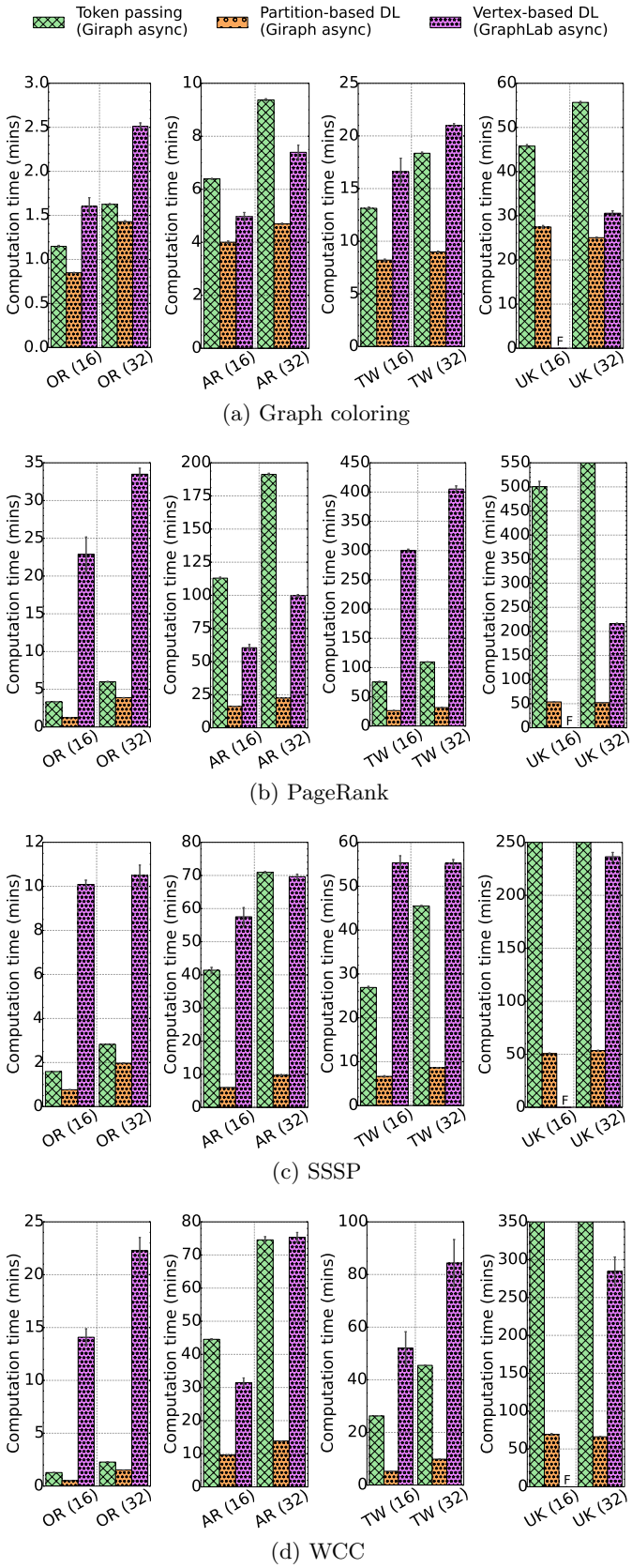
For our results, we report computation time, which is the total time of running an algorithm minus the input loading and output writing times. This also captures any communication overheads that the synchronization techniques may have: poor use of network resources translates to longer computation times.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

For graph coloring, partition-based locking is up to 2.3× faster than vertex-based locking for TW with 32 machines (Figure 9a). This is despite the fact that Giraph async performs an additional iteration compared to GraphLab async (Section 7.2.1). Similarly, partition-based locking is up to 2.2× faster than token passing for UK on 32 machines. Vertex-based locking fails for UK on 16 machines because GraphLab async runs out of memory.

As detailed in Section 5.4, these performance gains arise from significantly reducing the communication overheads, which is achieved by sharing fewer forks between larger partitions instead of millions or billions of forks between individual vertices. Moreover, unlike vertex-based locking, partition-based locking is able to support message batching, which further reduces communication overheads.

For PageRank, partition-based distributed locking again outperforms the other techniques: up to 18× faster than vertex-based locking on OR with 16 machines (Figure 9b). Vertex-based locking again fails for UK on 16 machines due to GraphLab async exhausting system memory. Token passing takes over 12 hours (720 mins) for UK on 32 machines due to its limited parallelism (Section 5.3), making partition-based locking over 14× faster than token passing.

For SSSP and WCC on UK, token passing takes over 7 hours (420 mins) for 16 machines and 9 hours (540 mins) for 32 machines, while GraphLab async fails on 16 machines due to running out of memory (Figures 9c and 9d). For SSSP, partition-based locking is up to 13× faster than vertex-based locking for OR on 16 machines and over 10× faster than token passing for UK with 32 machines. For WCC, partition-based locking is up to 26× faster than vertex-based locking for OR on 16 machines and over 8× faster than token passing for UK with 32 machines. These performance gains are larger because these algorithms, like many machine learning



(a) Graph coloring

(b) PageRank

(c) SSSP

(d) WCC

Figure 9: Computation times for graph coloring, PageRank, SSSP, and WCC. Missing bars are labelled with 'F' for unsuccessful runs.

algorithms, require multiple iterations to complete: the per-iteration performance gains, described earlier, are further multiplied by the number of iterations executed.

Partition-based distributed locking also scales better when going from 16 to 32 machines. For example, partition-based locking achieves a speedup with graph coloring on `UK`, whereas token passing suffers a slowdown (Figure 9a). In the cases where partition based-locking also experiences slowdown, which occurs because serializability trades off performance for stronger guarantees, its performance does not degrade as quickly as token passing and vertex-based locking and its computation time remains the shortest.

Lastly, Giraphx implements its synchronization techniques only for graph coloring, so we can compare against only this algorithm. As discussed previously, Giraphx implements its techniques as part of user algorithms rather than within the system, resulting in poor usability as they must be re-implemented in every user algorithm. A key advantage of our techniques is that, because they are implemented at the system level, serializability is automatically provided for all user algorithms. For graph coloring on `OR` with 16 machines, Giraphx with single-layer token passing is 30× and 41× slower than Giraph async with dual-layer token passing and partition-based distributed locking, respectively. With vertex-based locking, Giraphx is 55× slower than GraphLab async with vertex-based locking and 103× slower than Giraph async with partition-based locking. On `TW` and `UK`, Giraphx fails to run due to exhausting system memory. Giraphx's poor performance is due to its less efficient techniques, the fact that it uses a much older and less performant version of Giraph and, unlike Giraph async, does not implement the more performant version of the AP model.

8. RELATED WORK

To the best of our knowledge, this paper is the first to formulate the important notion of serializability for graph processing systems and to incorporate it into a foundational framework that has been implemented in a real system to deliver an end-to-end solution. Only Giraphx [33] and GraphLab [27, 18] provide serializability but, as we showed in this paper, our techniques significantly outperform their designs. Moreover, neither of their proposals provide a formal framework for reasoning about serializability nor do they show correctness for their synchronization techniques. Giraphx considers single-layer token passing and vertex-based distributed locking but its implementations are a part of user algorithms rather than within the system: each technique must be re-implemented in every user algorithm, which negatively impacts performance and usability. GraphLab async uses vertex-based distributed locking and is tailored for this synchronization technique. However, it is not partition aware and thus cannot support the more efficient partition-based distributed locking technique.

We mention several other vertex-centric graph processing systems next, however, they neither consider nor provide serializability. Apache Hama [2] is a general BSP system that, unlike Giraph, is not optimized for graph processing. GPS [31] and Mizan [25] are BSP systems that consider dynamic workload balancing, but not serializability, while GRACE [35] is a single-machine shared memory system that implements the AP model. GraphX [36] is a system built on the data parallel engine Spark [37], and considers graphs stored as tabular data and graph operations as distributed joins.

GraphX's primary goal is to provide efficient graph processing for end-to-end data analytic pipelines implemented in Spark. Pregelix [12] is a BSP graph processing system implemented in Hyracks [9], a shared-nothing dataflow engine. Pregelix stores graphs and messages as data tuples and uses joins to implement message passing. GraphChi [26] is a single-machine disk-based graph processing system for processing graphs that do not fit in memory.

9. CONCLUSION

We presented a formalization of serializability for graph processing systems and proved that two key conditions must hold to provide serializability. We then showed the need for partition aware synchronization techniques to provide serializability more efficiently. In particular, we introduced a novel partition-based distributed locking technique that, in addition to being correct, is more efficient than existing techniques. We implemented all techniques in Giraph to provide serializability as a configurable option that is completely transparent to algorithm developers. Our experimental evaluation demonstrated that our partition-based technique is up to 26× faster than existing techniques that are implemented by graph processing systems such as GraphLab.

10. REFERENCES

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

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

[3] GraphLab: Distributed Graph-Parallel API. `http://docs.graphlab.org/classgraphlab_1_1async_consistent__engine.html`, 2014.

[4] M. Balassi, R. Palovics, and A. A. Benczur. Distributed Frameworks for Alternating Least Squares. In *RecSys '14*.

[5] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, 1987.

[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*, pages 587–596, 2011.

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

[9] 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*, pages 1151–1162, 2011.

[10] A. Boularias, O. Krömer, and J. Peters. Structured Apprenticeship Learning. In *ECML PKDD '12*, pages 227–242, 2012.

[11] J. K. Bradley, A. Kyrola, D. Bickson, and C. Guestrin. Parallel Coordinate Descent for L1-Regularized Loss Minimization. In *ICML*, 2011.

[12] 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.

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

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

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

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

[17] J. Gonzalez, Y. Low, A. Gretton, and C. Guestrin. Parallel Gibbs Sampling: From Colored Fields to Thin Junction Trees. In *AISTATS*, volume 15, pages 324–332, 2011.

[18] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI '12*, pages 17–30.

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

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

[21] M. Han and K. Daudjee. Providing Serializability for Pregel-like Graph Processing Systems. *EDBT*, 2016.

[22] 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.

[23] 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.

[24] Karypis Lab. METIS and ParMETIS. `http://glaros.dtc.umn.edu/gkhome/views/metis`.

[25] 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.

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

[27] 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.

[28] 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*, pages 135–146, 2010.

[29] Mausam and A. Kolobov. *Planning with Markov Decision Processes: An AI Perspective*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

[30] D. Nicoara, S. Kamali, K. Daudjee, and L. Chen. Hermes: Dynamic Partitioning for Distributed Social Network Graph Databases. In *EDBT*, pages 25–36, 2015.

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

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

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

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

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

[36] 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.

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