Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing

ROBERT RYAN MCCUNE, TIM WENINGER, and GREG MADEY, University of Notre Dame

The vertex-centric programming model is an established computational paradigm recently incorporated into distributed processing frameworks to address challenges in large-scale graph processing. Billion-node graphs that exceed the memory capacity of commodity machines are not well supported by popular Big Data tools like MapReduce, which are notoriously poor performing for iterative graph algorithms such as PageRank. In response, a new type of framework challenges one to "think like a vertex" (TLAV) and implements user-defined programs from the perspective of a vertex rather than a graph. Such an approach improves locality, demonstrates linear scalability, and provides a natural way to express and compute many iterative graph algorithms. These frameworks are simple to program and widely applicable but, like an operating system, are composed of several intricate, interdependent components, of which a thorough understanding is necessary in order to elicit top performance at scale. To this end, the first comprehensive survey of TLAV frameworks is presented. In this survey, the vertex-centric approach to graph processing is overviewed, TLAV frameworks are deconstructed into four main components and respectively analyzed, and TLAV implementations are reviewed and categorized.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*; D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms*

General Terms: Design, Algorithms

Additional Key Words and Phrases: Graph processing, pregel, distributed systems, Big Data, distributed algorithms

ACM Reference Format:
Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. ACM Comput. Surv. 48, 2, Article 25 (October 2015), 39 pages.
DOI: http://dx.doi.org/10.1145/2818185

1. INTRODUCTION

The proliferation of mobile devices, ubiquity of the web, and plethora of sensors has led to an exponential increase in the amount data created, stored, managed, and processed. In March 2014, an IBM report claimed that 90% of the world's data had been generated in the last 2 years [Kim et al. 2014]. Big Data characterizes the problems faced by

This work was supported in part by the AFOSR Grant #FA9550-15-1-0003, as well as a Department of Education GAANN Fellowship awarded by the University of Notre Dame Department of Computer Science and Engineering.
Authors' addresses: R. R. McCune, 326 Cushing Hall, University of Notre Dame, Notre Dame, IN, 46656; email: rmccune@nd.edu; T. Weninger, 353 Fitzpatrick Hall, University of Notre Dame, IN, 46656; email: tweninge@nd.edu; G. Madey, 384 Fitzpatrick Hall, University of Notre Dame, Notre Dame, IN, 46556; email: gmadey@nd.edu.

conventional analytics systems with this dramatic expansion of data volume, velocity, and variety.

To address the challenges posed by Big Data, analytical systems are shifting from shared, centralized architectures to distributed, decentralized architectures. The MapReduce framework, and its open-source variant Hadoop, exemplifies this effort by introducing a programming model to facilitate efficient, distributed algorithm execution while abstracting away lower-level details [Dean and Ghemawat 2008]. Since inception, the Hadoop/MapReduce ecosystem has grown considerably in support of related Big Data tasks.

However, these distributed frameworks are not suited for all purposes, and in many cases can even result in poor performance [Munagala and Ranade 1999; Cohen 2009; Kang et al. 2009]. Algorithms that make use of multiple iterations, especially those using graph or matrix data representations, are particularly poorly suited for popular Big Data processing systems.

Graph computation is notoriously difficult to scale and parallelize, often due to inherent interdependencies within graph data [Lumsdaine et al. 2007]. As Big Data drives graph sizes beyond the memory capacity of a single machine, data must be partitioned to out-of-memory storage or distributed memory. However, for sequential graph algorithms, which require random access to all graph data, poor locality and the indivisibility of the graph structure cause time- and resource-intensive pointer chasing between storage mediums in order to access each datum.

In response to these shortcomings, new frameworks based on the *vertex-centric programming model* have been developed with the potential to transform the ways in which researchers and practitioners approach and solve certain problems [Malewicz et al. 2010]. Vertex-centric computing frameworks are platforms that iteratively execute a user-defined program over vertices of a graph. The user-defined vertex function typically includes data from adjacent vertices or incoming edges as input, and the resultant output is communicated along outgoing edges. Vertex program kernels are executed iteratively for a certain number of rounds, or until a convergence property is met. As opposed to the randomly accessible, "global" perspective of the data employed by conventional shared-memory sequential graph algorithms, vertex-centric frameworks employ a local, vertex-oriented perspective of computation, encouraging practitioners to "think like a vertex" (TLAV).

The first published TLAV framework was Google's Pregel system [Malewicz et al. 2010], which, based on Valiant's Bulk Synchronous Parallel (BSP) model [Valiant 1990], employs synchronous execution. While not all TLAV frameworks are synchronous, these frameworks are first introduced here within the context of BSP in order to provide foundational understanding of TLAV concepts.

1.1. Bulk Synchronous Parallel

After spending a year with Bill McColl at Oxford in 1988, Les Valiant published the seminal paper on the Bulk Synchronous Parallel (BSP) computing model [Valiant 1990] for guiding the design and implementation of parallel algorithms. Initially touted as "A Bridging Model for Parallel Computation," the BSP model was created to simplify the design of software for parallel hardware, thereby "bridging" the gap between high-level programming languages and multiprocessor systems.

As opposed to distributed shared memory or other distributed systems' abstractions, BSP makes heavy use of a message passing interface (MPI), which avoids high latency reads, deadlocks, and race conditions. BSP is, at the most basic level, a two-step process performed iteratively and synchronously: (1) perform task computation on local data and (2) communicate the results and then repeat the two steps. In BSP, each

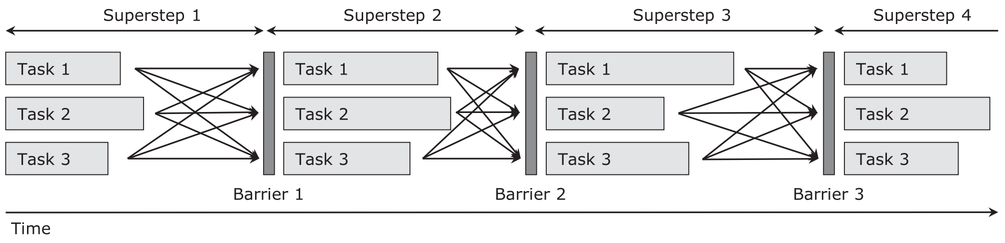


Fig. 1. Example of Bulk Synchronous Parallel execution with three tasks/workers over four supersteps. Each task may have varying durations after which messages are passed. The barriers control synchronization across the entire system.

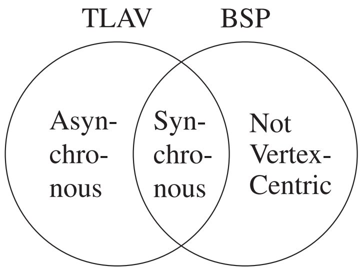


Fig. 2. Comparison of the Think Like a Vertex (TLAV) and Bulk Synchronous Parallel (BSP) models of computation. Both models are commonly employed for iterative computation.

compute/communicate iteration is called a *superstep*, with synchronization of the parallel tasks occurring at the superstep barriers, depicted in Figure 1.

1.2. Graph Parallel Systems

Introduced in 2010, the Pregel system [Malewicz et al. 2010] is a BSP implementation that provides an API specifically tailored for graph algorithms, challenging the programmer to “think like a vertex.” Graph algorithms are developed in terms of what each vertex has to compute based on local vertex data, as well as data from incident edges and adjacent vertices. The Pregel framework, as well as other synchronous TLAV implementations, split computation into BSP-style supersteps. Analogous to “components” in BSP [Valiant 1990], at each superstep a vertex can execute the user-defined vertex function and then send results to neighbors along graph edges. Supersteps always end with a synchronization barrier, shown in Figure 1, which guarantees that messages sent in a given superstep are received at the beginning of the next superstep. Unlike the original BSP model, vertices may change status between active and inactive, depending on the overall state of execution. Pregel terminates when all vertices halt and no more messages are exchanged.

A comparison of TLAV frameworks and BSP is presented in Figure 2. BSP employs a general model of broad applicability, including graph algorithms at varying levels of granularity. Underlying BSP execution is the global synchronization barrier among distributed processors. TLAV frameworks utilize a vertex-centric programming model, and while Pregel and its derivatives employ BSP-founded synchronous execution, other frameworks implement asynchronous execution, which has been demonstrated to improve performance in some instances [Xie et al. 2015].

In contrast to TLAV and BSP, MapReduce does not natively support iterative algorithms. Several recent frameworks have extended the MapReduce model to support iterative execution [Kajdanowicz et al. 2014], but for iterative graph algorithms, the graph topological data, which remains static, must be transferred from mappers to reducers, resulting in significant network overhead that renders iterative MapReduce frameworks uncompetitive with TLAV frameworks [Kajdanowicz et al. 2014]. A theoretical comparison between MapReduce and BSP is presented in Pace [2012].

1.3. TLAV Frameworks

Since Pregel, several TLAV frameworks have been proposed that either employ conceptually alternative framework components (such as asynchronous execution) or improve upon the Pregel model with various optimizations. This survey provides the first comprehensive examination into TLAV framework concepts and makes these other contributions:

(1) Analyzes four principle components in the design of vertex programs' execution in TLAV frameworks, identifying the tradeoffs in component implementations and providing data-driven discussion
(2) Overviews approaches related to TLAV system architecture, including fault tolerance on distributed systems and novel techniques for large-scale processing on single machines
(3) Discusses how the scalability of a graph algorithm varies inversely with the algorithm's scope, illustrated by vertex-centric and related subgraph-centric, or hybrid, frameworks

This article is organized as follows: First, Section 2 overviews the vertex-centric programming model, including an example program and execution. Section 3 presents the four major design decisions, or pillars, of the vertex-centric model. Section 4 presents details for distributed implementation, as well as novel techniques utilized by TLAV frameworks that enable large-scale graph processing on a single machine. Section 5 presents subgraph-centric, or hybrid, frameworks, that adopt a computational scope of the graph that is greater than a vertex (TLAV) but less than the entire graph. Section 6 discusses related work. Finally, Section 7 presents a summary, conclusions, and directions for future work.

First, a brief note on terminology: The TLAV paradigm is described interchangeably as *vertex centric*, *vertex oriented*, or *think like a vertex*. A *vertex program kernel* refers to an instance of the user-defined vertex *program*, *function*, or *process* that is executed on a particular vertex. A graph is a data structure made up of vertices and edges, both with (potentially empty) data properties. As in the literature, *graph* and *network* may be used interchangeably, as may *node* and *vertex*, and *edge* and *link*. *Network* may also refer to hardware connecting two or more machines, depending on context. A *worker* refers to a slave machine in the conventional master–worker architectural pattern, and a *worker process* is the program that governs worker behavior, including, but not limited to, execution of vertex programs, intermachine communication, termination, and check pointing. Graphs are assumed to be directed without loss of generality.

2. OVERVIEW

Graph processing is transitioning from centralized to decentralized design patterns. Sequential, shared-memory graph algorithms are inherently centralized. Conventional graph algorithms, such as Dijkstra's shortest path [Dijkstra 1971] or betweenness centrality [Freeman 1977], receive the entire graph as input and presume all data is randomly accessible in memory (i.e., graph-omniscient algorithms), and a centralized computational agent processes the graph in a sequential, top-down manner. However, the unprecedented size of Big Data-produced graphs, which may contain hundreds of billions of nodes and occupy terabytes of data or more, exceed the memory capacity of standard machines. Moreover, attempting to centrally compute graph algorithms across distributed memory results in unmanageable pointer chasing [Lumsdaine et al. 2007]. A more local, decentralized approach is required for processing graphs of scale.

Think-like-a-vertex frameworks are platforms that iteratively execute a user-defined program over vertices of a graph. The vertex program is designed from the perspective of a vertex, receiving as input the vertex's data as well as data from adjacent vertices and incident edges. The vertex program is executed across vertices of the graph synchronously or may also be executed asynchronously. Execution halts after either a specified number of iterations or all vertices have converged. The vertex-centric programming model is less expressive than conventional graph-omniscient algorithms but is easily scalable with more opportunity for parallelism.

The frameworks are founded in the field of distributed algorithms. Although vertex-centric algorithms are local and bottom up, they have a provable, global result. TLAV frameworks are heavily influenced by distributed algorithms theory, including synchronicity and communication mechanisms [Lynch 1996]. Several distributed algorithm implementations, such as distributed Bellman-Ford single-source shortest path [Lynch 1996], are used as benchmarks throughout the TLAV literature. The recent introduction of TLAV frameworks has also spurred the adaptation of many popular Machine Learning and Data Mining (MLDM) algorithms into graph representations for high-performance TLAV processing of large-scale datasets [Low et al. 2010].

Many graph problems can be solved by both a sequential, shared-memory algorithm and a distributed, vertex-centric algorithm. For example, the PageRank algorithm for calculating web page importance has a centralized matrix form [Page et al. 1999] as well as a distributed, vertex-centric form [Malewicz et al. 2010]. The existence of both forms illustrates that many problems can be solved in more than one way, by more than one approach or computational perspective, and deciding which approach to use depends on the task at hand. While the sequential, shared-memory approach is often more intuitive and easier to implement on a single machine or centralized architecture, the limits of such an approach are being reached.

Vertex programs, in contrast, only depend on data local to a vertex and reduce computational complexity by increasing communication between program kernels. As a result, TLAV frameworks are highly scalable and inherently parallel, with manageable intermachine communication. For example, runtime on the Pregel framework has been shown to scale linearly with the number of vertices on 300 machines [Malewicz et al. 2010]. Furthermore, TLAV frameworks provide a common interface for vertex program execution, abstracting away low-level details of distributed computation, like MPI, allowing for a fast, reusable development environment. A paradigm shift from centralized to decentralized approaches to problem solving is represented by TLAV frameworks.

2.1. Example: Single-Source Shortest Path in TLAV Paradigm

The following describes a simple vertex program that calculates the shortest paths from a given vertex to all other vertices in a graph. In contrast to this distributed implementation example, consider a centralized, sequential, shared-memory, or "graph-omniscient" solution to the single-source shortest path algorithm known as Djikstra's algorithm [Dijkstra 1959] or the more general BellmanFord algorithm [Bellman 1958].

Both the Dijkstra and the Bellman-Ford algorithms are based on repeated relaxations, which iteratively replace distance estimates with more accurate values until eventually reaching the solution. Both variants have a superlinear time complexity: Djisktra's runs in $O(|E|\log|E| + |V|)$ and Bellman-Ford's runs in $O(|E| \times |V|)$, where

ALGORITHM 1: Single-Source Shortest Path for a Synchronized TLAV Framework

```
  input: A graph $(V, E) = G$ with vertices $v \in V$ and edges from $i \to j$ s.t. $e_{ij} \in E$,
         and starting point vertex $v_s \in V$
  foreach $v \in V$ do shrtest_path_len$_v \leftarrow \infty$;     /* initialize each vertex data to $\infty$ */
  send (0, $v_s$);                        /* to activate, send msg of 0 to starting point */
  repeat               /* The outer loop is synchronized with BSP-styled barriers */
      for $v \in V$ do in parallel                       /* vertices execute in parallel */
          /* vertices inactive by default; activated when msg received         */
          /* compute minimum value received from incoming neighbors              */
1         minIncomingData$\leftarrow$ min(receive (path_length));
          /* set current vertex-data to minimum value                             */
2         if minIncomingData $<$ shrtest_path_len$_v$ then
3             shrtest_path_len$_v$ $\leftarrow$ minIncomingData;
4             foreach $e_{vj} \in E$ do
                  /* send shortest path + edge weight to outgoing edges           */
5                 path_length $\leftarrow$ shrtest_path_len$_v$+weight$_e$;
6                 send (path_length, $j$);
7             end
8         end
9         halt ();
      end
  until no more messages are sent;
```

$|E|$ is the number of edges and $|V|$ is the number of vertices in the graph, and typically $|E| \gg |V|$. Perhaps more importantly, both procedural, shared-memory algorithms keep a large state matrix resulting in a space complexity of $O(|V|^2)$.

In contrast, to solve the same single-source shortest path problem in the TLAV programming model, a vertex program need only pass the minimum value of its incoming edges to its outgoing edges during each superstep. This algorithm, considered a distributed version of Bellman-Ford [Lynch 1996], is shown in Algorithm 1. The computational complexity of each vertex program kernel is less than that of the sequential solution; however, a new dimension is introduced in terms of the communication complexity, or the messaging between vertices [Lynch 1996]. For TLAV implementation, a user need only write the inner portion of Algorithm 1 denoted by line numbers; the outermost loop and the parallel execution are handled by the framework. Because lines 1 through 10 are executed on the each vertex, these lines are known as the *vertex program*.

The TLAV solution to the single-source shortest path problem has surprisingly few lines of code, and understating its execution requires a different way of thinking.

Figure 3 depicts the execution of Algorithm 1 for a graph with four vertices and six weighted directed edges. Only the source vertex begins in an active state. In each superstep, a vertex processes its incoming messages and determines the smallest value among all messages received, and if the smallest received value is less than the vertex's current shortest path, then the vertex adopts the new value as its shortest path and sends the new path length plus respective edge weights to outgoing neighbors. If a vertex does not receive any new messages, then the vertex becomes inactive, represented as a shaded vertex in Figure 3. Overall execution halts once no more messages are sent and all vertices are inactive.

With this example providing insight into TLAV operation, particularly the synchronous message-passing model of Pregel, the survey continues by more completely detailing TLAV properties and categorizing different TLAV frameworks.

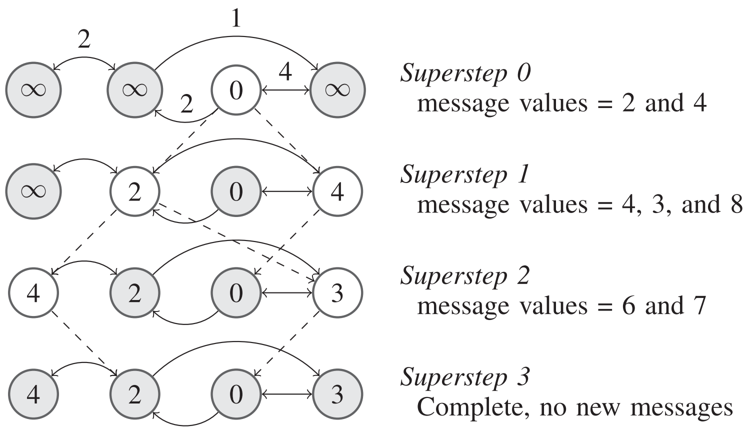


Fig. 3. Computing the single-source shortest path in a graph. Dashed lines between supersteps represent messages (with values listed to the right), and shaded vertices are inactive. Edge weights are pictorially included in the first layer for Superstep 0 and then subsequently omitted.

3. FOUR PILLARS OF TLAV FRAMEWORKS

A TLAV framework is software that supports the iterative execution of a user-defined vertex program over vertices of a graph. Frameworks are composed of several interdependent components that drive program execution and ultimate system performance. These frameworks are not unlike an analytic operating system, where component design decisions dictate how computations for a particular topology utilize the underlying hardware.

This section introduces the four principal pillars of TLAV frameworks. They are:

(1) Timing - How user-defined vertex programs are scheduled for execution
(2) Communication - How vertex program data is made accessible to other vertex programs
(3) Execution Model - Implementation of vertex program execution and flow of data
(4) Partitioning - How vertices of the graph, originally in storage, are divided up to be stored across memory of the system's multiple[1] worker machines

The discussion proceeds as follows: The timing policy of vertex programs is presented in Section 3.1, where system execution can be synchronous, asynchronous, or hybrid. Communication between vertex programs is presented in Section 3.2, where intermediate data is shared primarily through message passing or shared memory. The implementation of vertex program execution is presented in Section 3.3, which overviews popular models of program execution and demonstrates how a particular model implementation impacts execution and performance. Finally, partitioning of the graph from storage into distributed memory is presented in Section 3.4.

[1]TLAV systems generally distribute a graph across multiple machines because of the graph's prohibitive size. However, regarding the categorization of "single machine frameworks" in Section 4.4, while some TLAV frameworks are implemented for a single machine without the specific intention of developing for a nondistributed environment (e.g., the framework is first developed for a single machine before developing the framework for a distributed environment, like the original GraphLab [which published a distributed version 2 years later, see Section 3.1.2] or GRACE [see Section 3.1.3]), the single-machine frameworks presented in Section 4.4 are frameworks that implement particularly *novel* methods with the *stated objective* of processing, on a single machine, graphs of size that exceed the single machine's memory capacity. These single-machine frameworks still partition the graph, using framework-specific methods detailed in the respective section.

Table I. Execution Timing Model of Selected Frameworks

Framework	Timing	
Pregel	Synchronous	[Malewicz et al. 2010]
Giraph	Synchronous	[Avery 2011]
Hama	Synchronous	[Seo et al. 2010]
GraphLab	Asynchronous	[Low et al. 2012, 2010]
PowerGraph	Both	[Gonzalez et al. 2012]
PowerSwitch	Hybrid	[Xie et al. 2015]
GRACE	Hybrid	[Wang et al. 2013]
GraphHP	Hybrid	[Chen et al. 2014a]
P++	Hybrid	[Zhou et al. 2014]

Each pillar is heavily interdependent with other pillars, as each design decision is tightly integrated and strongly influenced by other design decisions. While each pillar may be understood through a sequential reading of the information provided, a more efficient yet thorough understanding may be achieved by freely forward- and cross-referencing other pillars, especially when related sections are cited. The interrelation of the four pillars is unavoidable and indivisible, *not unlike a graph data structure itself.* The difficulty of independently describing each pillar certainly reflects the challenge of processing a vertex in which a given result depends on the concurrent processing of neighboring vertices. This survey is restricted to a sequential presentation of information in the form of an article. However, each pillar, though unique, depends on, and may only be described in relation to, other pillars, so a sufficient understanding of any given pillar may only be achieved by understanding all pillars of a TLAV framework, collectively. Thus, one may begin to understand the challenges of processing graphs (especially large graphs, when not all "pillars" are in the same "paper") as in Section 1, Section 2, and Lumsdaine et al. [2007].

3.1. Timing

In TLAV frameworks, the scheduling and timing of the execution are separate from the logic of the vertex program. The *timing* of a framework characterizes how active vertices are ordered by the scheduler for computation. Timing can be synchronous, asynchronous, or a hybrid of the two models. Frameworks that represent the different fundamental timing models are presented in Table I.

3.1.1. Synchronous. The *synchronous* timing model is based on the original BSP processing model discussed earlier. In this model, active vertices are executed conceptually in parallel over one or more iterations, called *supersteps*. Synchronization is achieved through a global synchronization *barrier* situated between each superstep that blocks vertices from computing the next superstep until all workers complete the current superstep. Each worker coordinates with the master to progress to the next superstep. Synchronization is achieved because the barrier ensures that each vertex within a superstep has access to only the data from the previous superstep. Within a single processing unit, vertices can be scheduled in a fixed or random order because the execution order does not affect the state of the program. The global synchronization barrier introduces several performance tradeoffs.

Synchronous systems are conceptually simple, demonstrate scalability, and perform exceptionally well for certain classes of algorithms. While not all TLAV programs consistently converge to the same values depending on system implementation, synchronous systems are almost always deterministic, making synchronous applications easy to design, program, test, debug, and deploy. Although coordinating synchronization imposes consistent overhead, the overhead becomes largely amortized for large graphs. Synchronous systems demonstrate good scalability, with runtime often linearly increasing with the number of vertices [Malewicz et al. 2010]. As will be discussed in Section 3.2.1, synchronous systems are often implemented along with message-passing communication, which enables a more efficient "batch messaging" method. Batch messaging can especially benefit systems with lots of network traffic induced by algorithms with a low computation-to-communication ratio [Xie et al. 2015].

Although synchronous systems are conceptually straightforward and scale well, the model is not without drawbacks. One study found that synchronization, for an instance of finding the shortest path in a highly partitioned graph, accounted for over 80% of the total running time [Chen et al. 2014a], so system throughput must remain high to justify the cost of synchronization, since such coordination can be relatively costly. However, when the number of active vertices drops or the workload among workers becomes imbalanced, system resources can become underutilized. Iterative algorithms often suffer from "the curse of the last reducer," otherwise known as the "straggler" problem, where many computations finish quickly but a small fraction of computations take a disproportionately longer amount of time [Suri and Vassilvitskii 2011]. *For synchronous systems, each superstep takes as long as the slowest vertex*, so synchronous systems generally favor lightweight computations with small variability in runtime.

Finally, synchronous algorithms may not converge in some instances. In graph coloring algorithms, for example, vertices attempt to choose colors different from adjacent neighbors [Gonzalez et al. 2011] and require coordination between neighboring vertices. However, during synchronous execution, the circumstance may arise where two neighboring vertices continually flip between each other's color. In general, algorithms that require some type of neighbor coordination may not always converge with the synchronous timing model without the use of some extra logic in the vertex program [Xie et al. 2015].

3.1.2. Asynchronous. In the asynchronous iteration model, no explicit synchronization points (i.e., barriers) are provided, so any active vertex is eligible for computation whenever processor and network resources are available. Vertex execution order can be dynamically generated and reorganized by the scheduler, and the "straggler" problem is eliminated. As a result, many asynchronous models outperform corresponding synchronous models, but at the expense of added complexity.

Theoretical and empirical research has demonstrated that asynchronous execution can generally outperform synchronous execution [Bertsekas and Tsitsiklis 1989; Low et al. 2012], although precise comparisons for TLAV frameworks depend on a number of properties [Xie et al. 2015]. Asynchronous systems especially outperform synchronous systems when the workload is imbalanced. For example, when computation per vertex varies widely, synchronous systems must wait for the slowest computation to complete, while asynchronous systems can continue execution maintaining high throughput. One disadvantage, however, is that asynchronous execution cannot take advantage of batch messaging optimizations (see Section 3.2.4). Thus, synchronous execution generally accommodates I/O-bound algorithms, while asynchronous execution well-serves CPU-bound algorithms by adapting to large and variable workloads.

Many iterative algorithms exhibit asymmetric convergence. Low et al. [2012] demonstrated that for PageRank, the majority of vertices converged within one superstep, while only 3% of vertices required more than 10 supersteps. Asynchronous systems can utilize prioritized computation via a dynamic schedule to focus on more challenging computations early in execution to achieve better performance [Zhang et al. 2013; Low et al. 2012]. Generally, asynchronous systems perform well by providing more execution flexibility and by adapting to dynamic or variant workloads.

Although intelligent scheduling can improve performance, schedules resulting in suboptimal performance are also possible. In some instances, a vertex may perform more updates than necessary to reach convergence, resulting in excessive computation [Zhang et al. 2012a]. Moreover, if implementing the pull model of execution, which is commonly implemented in asynchronous systems [Low et al. 2012] and described in Section 3.3.2, communication becomes redundant when neighboring vertex values don't change [Zhang et al. 2012a; Hant et al. 2014].

The flexibility provided by asynchronous execution comes at the expense of added complexity, not only from scheduling logic, but also from maintaining data consistency. Asynchronous systems typically implement shared memory, discussed in Section 3.2.2, where data-race conditions can occur when parallel computations simultaneously attempt to modify the same data. Additional mechanisms are necessary to ensure mutual exclusion, which can challenge algorithm development because framework users may have to consider low-level concurrency issues [Wang et al. 2013], as, for example, in GraphLab, where users must select a consistency model [Low et al. 2012].

3.1.3. Hybrid. Rather than adhering to the inherent strengths and weaknesses of a strict execution model, several frameworks work around a particular shortcoming through design improvements. One such implementation, GraphHP, reduces the high fixed cost of the global synchronization barrier using *pseudo-supersteps* [Chen et al. 2014a]. Another implementation, GRACE, explores dynamic scheduling within a single superstep [Wang et al. 2013]. The PowerSwitch system removes the need to choose between synchronous and asynchronous execution and instead adaptively switches between the two modes to improve performance [Xie et al. 2015]. Together, these three frameworks illustrate how weaknesses with a particular execution model can be overcome through engineering and problem solving, rather than strict adoption of an execution model.

As previously discussed, synchronous systems suffer from the high, fixed cost of the global synchronization barrier. The hybrid execution model introduced by GraphHP, and also used by the P++ framework [Zhou et al. 2014], reduces the number of supersteps by decoupling intraprocessor computation from the interprocessor communication and synchronization [Chen et al. 2014a]. To do this, GraphHP distinguishes between two types of nodes: *boundary nodes* that share an edge across partitions, and *local nodes* that only have neighboring nodes within the local partition. During synchronization, messages are only exchanged between boundary nodes. As a result, in GraphHP, a given superstep is composed of two phases: global and local. The global phase, which is executed first, runs the user program across all boundary vertices using data transmitted from other boundary vertices as well as its own local vertices. Once the global phase is complete, the local phase executes the vertex program on local vertices within a pseudo-superstep; the pseudo-superstep is different from a regular superstep in that (1) pseudo-supersteps have local barriers resulting in local iterations independent of any global synchronization or communication, and (2) local message passing is done through direct, in-memory message passing, which is much faster than standard MPI-style messages.

A similar approach to segmented execution, as in GraphHP and P++, is the KLA paradigm [Harshvardhan et al. 2014], which creates a hybrid of synchronous and asynchronous execution. For graphs, the depth of asynchronous execution is parameterized, and asynchronous execution is allowed for a certain number of levels before a synchronous round. Similar to how GraphHP implements a round of boundary vertex execution before several rounds of local execution, KLA has multiple traversals of asynchronous execution before coordinating a round of synchronous execution. The tradeoff is between expensive global synchronizations with cheap but possibly redundant asynchronous computations. KLA is also similar to delta-stepping used for single-source shortest path [Meyer and Sanders 2003].

The single-machine framework GRACE explores dynamic scheduling of vertices from within a single synchronous round [Wang et al. 2013]. To do this, GRACE exposes a programming interface that, from within a given superstep, allows for prioritized execution of vertices and selective receiving of messages outside of the previous superstep. Results demonstrate comparable runtime to asynchronous models, with better scaling across multiple worker threads on a single machine.

Knowing a priori which execution mode will perform better for a given problem, algorithm, system, or circumstance is challenging. Furthermore, the underlying properties that give one execution model an advantage over another may change over the course of processing. For example, in the distributed single-source shortest path algorithm [Bertsekas et al. 1996], the process begins with few active vertices, where asynchronous execution is advantageous, then propagates to a high number of active vertices performing lightweight computations, which is ideal for synchronous execution, before finally converging among few active vertices [Xie et al. 2015]. For some algorithms, one execution mode may outperform another only for certain stages of processing, and the best mode at each stage can be difficult to predict.

Motivated by the necessity for execution mode dynamism, PowerSwitch was developed to adaptively switch between synchronous and asynchronous execution modes [Xie et al. 2015]. Developed on top of the PowerGraph platform, PowerSwitch can quickly and efficiently switch between synchronous and asynchronous execution. PowerSwitch incorporates throughput heuristics with online sampling to predict which execution mode will perform better for the current period of computation. Results demonstrate that PowerSwitch's heuristics can accurately predict throughput, the switching between the two execution modes is well timed, and overall runtime is improved for a variety of algorithms and system configurations [Xie et al. 2015].

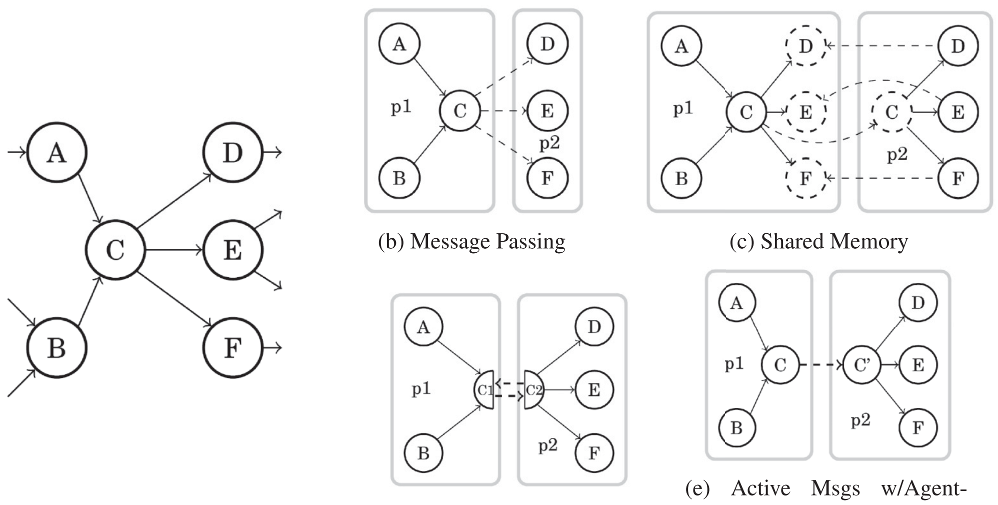
3.2. Communication

Communication in TLAV frameworks entails how data is shared between vertex programs. The two conventional models for communication in distributed systems, as well as distributed algorithms, are message passing and shared memory [Yan et al. 2014b; Lu et al. 1995; Lynch 1996]. In message-passing systems, data is exchanged between processes through messages, whereas in shared-memory systems, data for one process is directly and immediately accessible by another process. This section compares and contrasts message passing and shared memory for TLAV frameworks. A third method of communication, active messages, is also presented. Finally, techniques to optimize distributed message passing are discussed.

Diagrams in Figure 4 are referenced throughout this section to illustrate the different communication implementations. A sample graph is presented in Figure 4(a), and Figures 4(b) to 4(e) depict four TLAV communication implementations of the sample graph. For each implementation, vertices are partitioned across two machines, namely, vertices A, B, and C are partitioned to machine p1, and vertices D, E, and F are put on machine p2 (except Figure 4(d) and 4(e), where the graph is cut along vertex C). Solid arrows represent local communication,[2] and dashed arrows represent network traffic.

3.2.1. Message Passing. In the message-passing method of communication, also known as the LOCAL model of distributed computation [Peleg 2000], information is sent from one vertex program kernel to another via a message. A message contains local vertex data and is addressed to the ID of the recipient vertex. In the archetypal message-passing framework Pregel [Malewicz et al. 2010], a message can be addressed

[2]Local communication means communication between vertices residing on the same machine.



(a) Sample Graph (b) Message Passing (c) Shared Memory (d) Shared Mem w/Vertex-Cuts (e) Active Msgs w/Agent-Graph Scatter Vertex

Fig. 4. Distributed communication patterns for common communication implementations. The sample graph is partitioned across two machines (see Section 3.4), with vertices A, B, and C residing on machine p1, and vertices D, E, and F on machine p2. Pregel is represented in (b), GraphLab in (c), PowerGraph in (d), and GRE in (e).

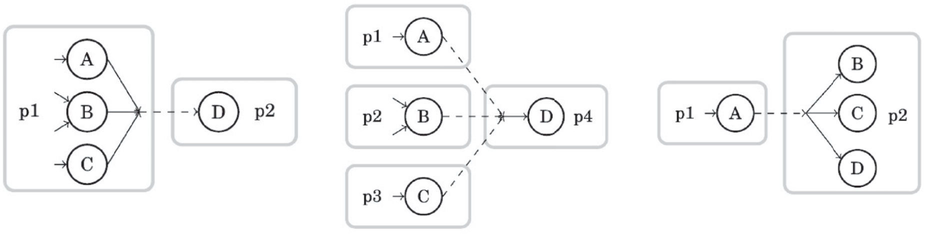
anywhere, but because vertices do not have ID information of all of the other vertices, destination vertex IDs are typically obtained by iterating over outgoing edges.

After computation is complete and a destination ID for each message is determined, the vertex dispatches messages to the local worker process. The worker process determines whether the recipient resides on the local machine or a remote machine. In the case of the former, the worker process can place the message directly into the vertex's incoming message queue. Otherwise, the worker process looks up the worker ID of the destination vertex[3] and places the message in an outgoing message buffer. The outgoing message buffer in Pregel, a synchronously timed system, is flushed when it reaches a certain capacity, sending messages over the network in batches. Waiting until the end of a superstep to send all outgoing remote messages can exceed memory limits [Satish et al. 2014].

Message passing is commonly implemented with synchronized execution, which guarantees data consistency without low-level implementation details. All messages sent during superstep S are received in superstep $S + 1$, at which point a vertex program can access the incoming message queue at the beginning of $S + 1$'s program execution. Synchronous execution also facilitates batch messaging, which improves network throughput. For I/O-bound algorithms with lightweight computation, such as PageRank [Brin and Page 1998], where vertices are "always active" so messaging is high [Shang and Yu 2013], synchronous execution has been shown to significantly outperform asynchronous execution [Xie et al. 2015].

Message passing is depicted in Figure 4(b), where vertex C sends (an) intermachine message(s) to vertices D, E, and F. Technically, messages are first sent from C to the worker process of *p1*, which routes the messages to worker process *p2*, which places the message in a vertex's incoming message queue, but the worker-process-related

[3]If the graph is partitioned using random hash partitioning, then the destination worker can be determined by hashing the destination vertex ID. Otherwise, for more advanced partitioning methods (see Section 3.4), the worker process typically has access to a local routing table, provided by the master during initialization.



(a) Sender-side Combiner (b) Receiver-side Combiner (c) Receiver-side Scatter

Fig. 5. Partition-driven optimization strategies for distributed message passing. The Combiner technique employs both Sender-side and Receiver-side Combiners.

routing is omitted from the figure without loss of generality. Figure 4(b) represents a general message-passing framework, such as Pregel or Giraph. The three messages sent by C across the network can be potentially reduced using optimization techniques in Section 3.2.4, namely, Receiver-side Scatter, depicted in Figure 5(c).

3.2.2. Shared Memory. Shared memory exposes vertex data as shared variables that can be directly read or be modified by other vertex programs. Shared memory avoids the additional memory overhead constituted by messages and doesn't require intermediate processing by workers. Shared memory is often implemented by TLAV frameworks developed for a single machine (see Section 4.4), since challenges to a shared-memory implementation arise in the distributed setting [Protic et al. 1997; Nitzberg and Lo 1991], where consistency must be guaranteed for remotely accessed vertices. Intermachine communication for distributed shared memory still occurs through network messages. The Trinity framework [Shao et al. 2013] implements a shared global address space that abstracts away distributed memory.

For shared-memory TLAV frameworks, race conditions may arise when an adjacent vertex resides on a remote machine. Shared-memory TLAV frameworks often ensure memory consistency through mutual exclusion by requiring serializable schedules. Serializability, in this case, means that every parallel execution has a corresponding sequential execution that maintains consistency, *cf.*, the dining philosophers problem [Low et al. 2012; Gonzalez et al. 2012].

In GraphLab [Low et al. 2012], border vertices are provided locally cached *ghost* copies of remote neighbors, where consistency between ghosts and the original vertex is maintained using pipelined distributed locking [Dijkstra 1971]. In PowerGraph [Gonzalez et al. 2012], the second generation of GraphLab, graphs are partitioned by edges and cut along vertices (see vertex cuts in Section 3.4), where consistency across cached *mirrors* of the cut vertex is maintained using parallel Chandy-Misra locking [Chandy and Misra 1984]. GiraphX is a Giraph derivative with a synchronous shared-memory implementation [Tasci and Demirbas 2013], which again provides serialization through Chandy-Misra locking of border vertices, although without local cached copies. The reduced overhead of shared memory compared to message passing is demonstrated by GiraphX, which converges 35% faster than Giraph when computing PageRank on a large Web Graph [Tasci and Demirbas 2013]. Moreover, some iterative algorithms perform better under serialized conditions, such as Dynamic ALS [Zhou et al. 2008; Low et al. 2012] and popular Gibbs sampling algorithms that actually require serializability for correctness [Gonzalez et al. 2011].

Shared-memory implementations are depicted in Figure 4(c) and Figure 4(d). In Figure 4(c), ghost vertices, represented by dashed circles, are created for every neighboring vertex residing on a remote machine, as implemented by GraphLab [Low et al.

2012]. One disadvantage of shared-memory frameworks is seen when computing on scale-free graphs, which have a certain percentage of high-degree vertices, such as vertex C. In these cases, the graph can be difficult to partition [Leskovec et al. 2009], resulting in many ghost vertices.

Figure 4(d) depicts shared memory with vertex cuts as implemented by PowerGraph [Gonzalez et al. 2012]. PowerGraph combines vertex cuts (discussed in Section 3.4) with the three-phase Gather-Apply-Scatter computational model (see Section 3.3.1) to improve processing of scale-free graphs. In Figure 4(d), the graph is cut along vertex C, where $C1$ is arbitrarily chosen as the master and $C2$ as the mirror. For each iteration, a distributed vertex preforms computation where (1) both $C1$ and $C2$ compute a partial result based on local neighbors, (2) the partial result is sent over the network from the mirror $C2$ to the master $C1$, (3) the master computes the final result for the iteration, (4) the master transmits the result back to the mirror over the network, and then (5) the result is sent to local neighbors as necessary. PowerGraph demonstrates how the combination of advanced components, that is, vertex cuts and three-phase computation, can overcome processing challenges like imbalances arising from high-degree vertices in scale-free graphs.

Shared-memory systems are often implemented with asynchronous execution. Although consistency is fundamentally maintained in synchronous message-passing frameworks like Pregel, asynchronous, shared-memory frameworks like GraphLab may execute faster because of prioritized execution and low communication overhead, but at the expense of added complexity for scheduling and maintaining consistency. The added complexity challenges scalability, for as the number of machines and partitions increase, more time and resources become devoted to locking protocols.

Dynamic computation addresses asymmetric convergence by only updating necessary vertices. Shared memory with asynchronous execution is an effective platform for dynamic computation, because the movement of data is separated from computation, allowing vertices to access neighboring values even if the values haven't changed between iterations. This implies the *pull* mode of information flow; see Section 3.3.2. In contrast, a vertex in a message-passing framework would need all neighboring values delivered in order to perform an update, even if some values had not changed. Dynamic computation is possible with message passing in the Cyclops framework, which implements a *distributed immutable view*. Cyclops is a synchronous shared-memory framework [Chen et al. 2014b], where one of the replicated vertices is designated the master, which computes updates and messages the updated state to replicas at the end of an iteration. Cyclops outperforms synchronous message-passing frameworks by reducing the amount of processing performed by each worker parsing messages and is comparable to PowerGraph by delivering significantly fewer messages.

Significant deterioration in performance was noted in Han et al. [2014] and Lu et al. [2014] for larger graphs, although admittedly performance largely depends on algorithm behavior [Xie et al. 2015; Shang and Yu 2013]. In short, asynchronous shared-memory systems can potentially outperform synchronous message-passing systems, though the latter often demonstrate better scalability and generalization.

3.2.3. Active Messages. While message passing and shared memory are the two most commonly implemented forms of communication in distributed systems, a third method called *active messages* is implemented in the GRE framework [Yan et al. 2014b].[4] Active messaging is a way of bringing computation to data, where a message contains both data and the operator to be applied to the data [von Eicken et al. 1992]. Active messages

[4]Active messages as described in this section are different from message-passing messages that activate a vertex, as in Pregel.

are sent asynchronously and executed upon receipt by the destination vertex. Within the GRE architecture, active messages combine the process of sending and receiving messages, removing the need to store an intermediate state, like message queues or edge data. When combined with the framework's novel Agent-Graph model, described later, GRE demonstrates a 20% to 55% reduction in runtime compared to PowerGraph across three benchmark algorithms in real and synthetic datasets, including a 39% reduction in the execution time per iteration for PageRank on the Twitter graph when scaled across 192 cores over 16 machines when compared to a PowerGraph implementation on 512 cores across 64 machines [Yan et al. 2014b].

The GRE framework modifies the data graph into an Agent-Graph. The Agent-Graph is a model used internally by the framework but is not accessible to the user. The Agent-Graph adds *combiner* and *scatter* vertices to the original graph in order to reduce intermachine messaging. Figure 4(e) shows that an extra *scatter* vertex, C', is added to create the internal Agent-Graph model. The C' vertex acts as a Receiver-side Scatter depicted in Figure 5(c). This is useful because the new C' vertex allows C to only send one message across the network, which C' then disperses to vertices D, E, and F. Combiner vertices are also added to the Agent-Graph in the same way as Server-side Aggregation depicted in Figure 5(a). The Agent-Graph employed by GRE is similar to vertex cuts in PowerGraph except that GRE messaging is unidirectional, and active messages are also utilized for parallel graph computation in the Active Pebbles framework [Willcock et al. 2011; Edmonds 2013].

3.2.4. Message-Passing Optimizations. Message passing can be costly, especially over a network. Thus, several message-reducing strategies have been developed in order to improve performance. Some strategies are topology driven and, as such, exploit the graph layout across machines, while other techniques are applied to specific algorithmic behavior. Three topology-driven optimizations are depicted in Figure 5 for messaging between machines $p1$ and $p2$ (or messaging from $p1$, $p2$, and $p3$ to $p4$, for Figure 5(b)).

The Combiner, inspired by the MapReduce function of the same name [Dean and Ghemawat 2008], is a message-passing optimization originally used by Pregel [Malewicz et al. 2010]. Presuming the commutative and associative properties of a vertex function, a Combiner executes on a worker process and combines many messages destined for the same vertex into a single message. For example, if a vertex function computes the sum of all incoming messages, then a Combiner would detect all messages destined for a vertex v, compute the sum of the messages, and then send the new sum to v. A Combiner can especially reduce network traffic when v is remote, shown as *sender-side aggregation* (Figure 5(a)). When v is local, a combiner can still reduce memory overhead by aggregating messages before placement into the incoming message queue, shown as *receiver-side aggregation* (Figure 5(b)). For the single-source shortest path algorithm, a combiner implementation resulted in a fourfold reduction in network traffic [Malewicz et al. 2010].

A related technique is the *receiver-side scatter*. For instances where the same message is sent to multiple vertices on the same remote machine, network traffic can be reduced by sending only one message and then having the destination worker distribute multiple copies, depicted in Figure 5(c). The strategy has been employed in multiple frameworks, including the *Large Adjacency List Partitioning* in GPS [Salihoglu and Widom 2013] and IBM's X-Pregel [Bao and Suzumura 2013], as the *fetch-once* behavior in LFGraph [Hoque and Gupta Hoque and Gupta], and through *scatter* nodes of the Agent-Graph in GRE [Yan et al. 2014b]. The technique reduces network traffic by increasing memory and processing overhead, as worker nodes must store the outgoing adjacency lists of other workers. With this in mind, GPS maintains a threshold where receiver-side scatter would only be applied for vertices above a certain degree. Experiments showed that as the threshold is lowered, network traffic at first decreases then plateaus, while runtime decreases but then increases, demonstrating the existence of an optimal vertex-degree threshold. In X-Pregel, a 10-fold reduction in network traffic from Receiver-side Scatter resulted in a 1.5 times speedup [Bao and Suzumura 2013]. Clearly, the receiver-side scatter strategy can be effective, but unlike the combiner, it is not guaranteed to improve performance.

The three partition-driven optimizations in Figure 5 are related to the messaging structure of a framework, and not specific to algorithm behavior, although some assumptions are made regarding message computation. Computation for the combiner must be commutative and associative because order cannot be guaranteed, while messages for the receiver-side scatter must be identical and independent of the adjacency list. Still, the techniques are oriented around partition-level messaging and apply to the worker process, only requiring certain operational properties in order to work. The Message-Online-Computing model proposed in Zhou et al. [2014], which improves memory usage by processing messages in the queue as they are delivered, also requires operations be commutative.

Conversely, algorithm-specific message optimizations have also been developed that restructure vertex messaging patterns for certain algorithmic behaviors [Salihoglu and Widom 2014; Quick et al. 2012]. For algorithms that combine vertices into a supervertex, like Boruvka's Minimum Spanning Tree [Chung and Condon 1996], the Storing Edges at Subvertices (SEAS) optimization implements a subroutine where each vertex tracks its parent supervertex instead of sending adjacency lists [Salihoglu and Widom 2014]. For algorithms where vertices remove edges, as in the 1/2-approximation for maximum weight matching [Preis 1999], the Edge Cleaning on Demand (ECOD) optimization only deletes stale edges when, counterintuitively, activity is requested for the stale edge [Salihoglu and Widom 2014]. To avoid slow convergence, ECOD is only employed above a certain threshold, for example, when more than 1% of all vertices are active. Both SEAS and ECOD exploit a tradeoff between sending messages proportional to the number of vertices or proportional to the number of edges. Other strategies for reducing communication, based on *aggregate computation*, are discussed in Section 5.3.

3.3. Execution Model

The model of execution for vertex-centric programs describes the implementation of the vertex function and how data moves during computation.

3.3.1. Vertex Program Implementation. Vertex functions have been implemented as one-, two-, or three-phase models. Vertex functions have also been implemented as edge-centric functions. While the model choice does not typically impact the accuracy of the final result, combining certain implementations with other TLAV components can yield improved system performance for certain graph characteristics.

One Phase. The vertex programming abstraction implemented as a single function is well characterized by the Pregel framework [Malewicz et al. 2010]. The single compute function of a vertex object follows the general sequence of accessing input data, computing a new vertex value, and distributing the update. In a typical Pregel program, the input data is accessed by iterating through the input message queue (messages that may have utilized a combiner), applying an update function based on received data, and then sending the new value through messages addressed by iterating over outgoing edges. Details based on other design decisions may vary; for example, input and output data may be distributed through incident edges, or neighboring vertex data may be directly accessible, but in one-phase models the general sequence of vertex execution is performed within a single, programmed function. The `Vertex.Compute()` function is implemented in several TLAV frameworks in addition to Pregel, including its open-source implementations [Avery 2011; Seo et al. 2010] and several related variants [Salihoglu and Widom 2013; Bao and Suzumura 2013; Redekopp et al. 2013]. The one-phase function implementation is conceptually straight-forward, but other frameworks provide opportunities for improvement by dividing up the computation.

Two Phase. A two-phase vertex-oriented programming model breaks up vertex programming into two functions, most commonly referred to as the Scatter-Gather model. In Scatter-Gather, the *scatter* phase distributes a vertex value to neighbors, and the *gather* phase collects the inputs and applies the vertex update. While most single-phase frameworks (e.g., Pregel) can be converted into two phases, the Scatter-Gather model was first explicitly put forward in the Signal/Collect framework [Stutz et al. 2010]. The two-phase model is also presented as Scatter-Gather in Roy et al. [2013] and is presented as the Iterative Vertex-Centric (IVEC) programming model in Yoneki and Roy [2013]. The Scatter-Gather programming model commonly occurs in TLAV systems where data is read/written to/from edges.

Ligra and Polymer are frameworks implemented for single machines (see Section 4.4) that both implement a two-phase model. The user provides two functions, one function that executes across each vertex in the active subset and another function that executes all outgoing edges in the subset. The frameworks adopt a vertex-subset-centric programming model, which is similar to vertex centric, but the framework retains a centralized view of the graph, where the whole graph is within the scope of computation, which is possible because the entire graph resides on a single machine in this case. The two-phase model is executed within a program processing the whole graph.

A related two-phase programming model for message passing called Scatter-Combine is implemented in the GRE framework [Yan et al. 2014b]. This model utilizes active messages, which are messages that include both data and the operator to be executed on the data [von Eicken et al. 1992]. In the first phase of the model, messages are sent (Scattered) and the operators in the messages are executed (Combined) at the destination vertex. In the second phase, the combined result is used to update the vertex value. The Scatter-Combine model incorporates two phases differently than Scatter-Gather. Instead of the two-phase Scatter-Gather model of (1) Gather-Apply and (2) Scatter, the Scatter-Combine model uses active messages to institute (1) Scatter-Gather and then (2) Apply. The GRE framework combines Scatter-Combine with a novel representation of the underlying data graph, called the Agent-Graph, described earlier, to reduce communication and improve scalability for processing graphs with scale-free degree distributions.

Three Phase. A three-phase programming model is introduced in PowerGraph as the Gather-Apply-Scatter (GAS) model [Gonzalez et al. 2012]. The *Gather* phase performs a generic summation over all input vertices and/or edges, like a commutative associative combiner. The result is used in the *Apply* phase, which updates the central vertex value. The *Scatter* phase distributes the update by writing the value to the output edges. PowerGraph incorporates the GAS model with vertex-cut partitioning (see Section 3.4.3) to improve processing of power-law graphs.

Edge Centric. The X-Stream framework provides an *edge-centric* two-phase Scatter-Gather programming model [Roy et al. 2013], as opposed to a vertex-centric programming model. The model is edge centric because the framework iterates over edges of the graph instead of vertices. However, the framework may still be considered TLAV because the two-phase program operates on source and target vertices, adopting a similar local scope. X-Stream leverages streaming edge data instead of random access for efficient large-scale graph processing on a single machine and is discussed in Section 4.4 in further detail.

3.3.2. Push Versus Pull. The flow of information for vertex programs can be characterized as data being *pushed* or *pulled* [Nguyen et al. 2013; Hant et al. 2014; Cheng et al. 2012]. In *push* mode, information flows from the active vertex performing the update outward to neighboring vertices, as in Pregel-like message passing. In *pull* mode, information flows from neighboring vertices inward to the active vertex, as in GraphLab-like shared memory, when an active vertex reads neighbors' data. Few TLAV frameworks explicitly adopt a push or pull mode. Instead, the information flow arises from other design decisions. Still, analyzing a system as push or pull allows one to reason about other system properties. For example, asynchronous execution is supported by both modes, but sender-side combining is only possible in push mode [Cheng et al. 2012].

Push and pull modes are more commonly associated with databases and transactional processing, though they have been more explicitly incorporated in broader graph engines and temporal frameworks (see Section 6 for related work). The Galois framework, with a flexible computation model enabling the implementation of a vertex-centric interface, allows users to choose push or pull mode [Kulkarni et al. 2009; Nguyen et al. 2013], as does Kineograph [Cheng et al. 2012]. Chronos experiments with how push and pull modes impact caching [Hant et al. 2014].

Ligra is a single-machine graph-processing framework that dynamically switches between push- and pull-based operators based on a threshold. The framework is in part inspired by a recently developed shared-memory breadth-first search algorithm that achieves remarkable performance by switching between push and pull modes of exploration [Beamer et al. 2013]. This algorithm, Ligra, and PowerSwitch from Section 3.1.3 exemplify how performance can be improved by dynamically adapting the processing technique to properties of the graph.

The delta-caching optimization, which is introduced in PowerGraph [Gonzalez et al. 2012], reduces the pulling of redundant data by tracking value changes. In a three-phase model, an accumulator value is the result of the gather step. With delta-caching, a cached copy of the accumulator for each vertex is stored by the worker, requiring additional storage. If, for a given update, the change in the accumulator is minimal, then neighboring vertices aren't activated, and any change can be applied to the cached copy stored by the worker. A neighboring vertex can then use the cached copy during an update. For delta-caching to be available, the apply function must be commutative and associative and have an inverse function. Delta-caching reduces redundant pulling by not activating neighboring vertices for small changes and resulted in a 45% decrease in runtime for computing PageRank on the Twitter graph [Gonzalez et al. 2012].

3.4. Partitioning

Large-scale graphs must be divided into parts to be placed in distributed memory. Good partitions often lead to improved performance [Salihoglu and Widom 2013], but expensive strategies can end up dominating processing time, leading many implementations to incorporate simple strategies, such as random placement [Jain et al. 2013]. Effective partitioning evenly distributed the vertices for balanced workload while minimizing interpartition edges to avoid costly network traffic, a problem formally known as *k-way graph partitioning* that is NP-complete with no fixed-factor approximation [Andreev and Racke 2006; Meyerhenke et al. 2014].

Leading work in graph partitioning can be broadly characterized as (1) rigorous but impractical mathematical strategies or (2) pragmatic heuristics used in practice [Tsourakakis et al. 2014]. Practical strategies, such as those employed in the suite of algorithms known as METIS [Karypis and Kumar 1995], often employ a three-phase multilevel partitioning approach [Abou-Rjeili and Karypis 2006]. Partition size is often allowed to deviate in the form of a "slackness" parameter in exchange for better cuts [Karypis and Kumar 1996].

Graph partitioning with METIS partitioning software is often considered the de facto standard for near-optimal partitioning in TLAV frameworks [Stanton and Kliot 2012]. Despite a lengthy preprocessing time, METIS algorithms significantly reduce total communication and improve overall runtime for TLAV processing on smaller graphs [Salihoglu and Widom 2013]. However, for graphs of even medium size, the high computational cost and necessary random access of the entire graph render METIS and related heuristics impractical. Alternatives for large-scale graph partitioning include distributed heuristics presented in Section 3.4.1, streaming algorithms in Section 3.4.2, vertex cuts in Section 3.4.3, and dynamic repartitioning in Section 3.4.4.

3.4.1. Distributed Heuristics. Distributed heuristics are decentralized methods, requiring little or no centralized coordination. Distributed partitioning is related to distributed community detection in networks [Gehweiler and Meyerhenke 2010; Ramaswamy et al. 2005], the two main differences being that (1) communities can overlap whereas partitions cannot and (2) partitioning requires a priori specification of the number of partitions, whereas community detection typically does not. Much distributed partitioning work has been inspired by distributed community detection, namely, label propagation [Raghavan et al. 2007].

Label propagation occurs at the vertex level, where each vertex adopts the label of the plurality of its neighbors. Though the process is decentralized, label propagation for partitioning necessitates a varying amount of centralized coordination in order to maintain balanced partitions and prevent “densification”: a cascading phenomenon where one label becomes the overwhelming preference [Raghavan et al. 2007]. The densification problem is addressed in Vaquero et al. [2014], wherein a simple capacity constraint is enforced that is equal to the available capacity of the local worker divided by the number of nonlocal workers. In Ugander and Backstrom [2013], balanced vertex distribution is maintained by constraining label propagation and solving a linear programming optimization problem that maximizes a relocation utility function. In Rahimian et al. [2013], vertices swap labels, either with a neighbor or possibly a random node, and simulated annealing is employed to escape local optima. The cost of centralized coordination incurred by these methods is much less than the cost of random vertex access on a distributed architecture, as with ParMETIS.

More advanced label propagation schemes for partitioning are presented in Wang et al. [2014] and Slota et al. [2014]. In Wang et al. [2014], label propagation is used as the coarsening phase of a multilevel partitioning scheme, which processes the partitioning in blocks to accommodate multilevel partitioning for large-scale graphs. In Slota et al. [2014], several stages of label propagation are utilized to satisfy multiple partitioning objectives under multiple constraints. Zeng et al. [2012] use a parallel multilevel partitioning algorithm for k-way balanced graphs that operates in two phases: an aggregate phase that uses weighted label propagation, and then a partition phase that performs the stepwise minimizing RatioCut method.

3.4.2. Streaming. Streaming partitioning is a form of online processing that partitions a graph in a single pass. For TLAV frameworks, streaming partitioning is especially efficient since the partitioning can be performed by the graph loader, which loads the graph from disk onto the cluster. The accepted streaming model assumes a single, centralized graph loader that reads data serially from disk and chooses where to place the data among available workers [Stanton and Kliot 2012; Tsourakakis et al. 2014]. Centralized streaming heuristics can be adapted to run in parallel [Stanton and Kliot 2012]; however, depending on the heuristic, concurrency between the parallel partitioners would likely be required [Nishimura and Ugander 2013]. One of the first online heuristics was presented by Kernighan and Lin and is used as a subroutine in METIS [Kernighan and Lin 1970]. GraphBuilder [Jain et al. 2013] is a similar library that, in addition to partitioning, supports an extensive variety of graph-loading-related processing tasks.

A streaming partitioner on a graph loader reads data serially from disk, receiving one vertex at a time along with its neighboring vertices. In a single look at the vertex, the streaming partitioner must decide the final placement for the vertex on a worker partition, but the streaming partitioner has access to the entire subgraph of already placed vertices. In a variant of the streaming model, the partitioner has an available storage buffer with a capacity equal to that of a worker partition, so the partitioner may temporarily store a vertex and decide the partitioning later [Stanton and Kliot 2012]; however, this buffer is not utilized by the top-performing streaming partitioners. For most heuristics, the placement of later vertices is dependent on placement of earlier vertices, so the presentation order of vertices can impact the partitioning. Thus, an adverse ordering can drastically subvert partitioning efforts; however, experiments demonstrate that performance remains relatively consistent for breadth-first, depth-first, and random orderings of a graph [Stanton and Kliot 2012; Tsourakakis et al. 2014].

Two top-performing streaming partitioning algorithms are greedy heuristics. The first is linear deterministic greedy (LDG), a heuristic that assigns a vertex to the partition with which it shares the most edges while weighted by a penalty function linearly associated with a partition's remaining capacity. The LDG heuristic is presented in Stanton and Kliot [2012], where 16 streaming partitioning heuristics are evaluated across 21 different datasets. The use of a buffer in addition to the LDF heuristic has been adapted for streaming partitioning of massive Resource Description Framework (RDF) data [Wang and Chiu 2013]. Another variant uses *unweighted* deterministic greedy instead of LDG to perform greedy selection based on neighbors without any penalty function; this unweighted variant has been employed for distributed matrix factorization [Ahmed et al. 2013]. Further analysis of LDG-related heuristics on random graphs, as well as lower-bound proofs for random and adversarial stream ordering, is presented in Stanton [2014].

Another top-performing streaming partitioner is FENNEL [Tsourakakis et al. 2014], which is inspired by a generalization of optimal quasi-cliques [Tsourakakis et al. 2013]. FENNEL achieves high-quality partitions that are in some instances comparable with near-optimal METIS partitions. Both FENNEL and LDG have been adapted to the restreaming graph partitioning model, where a streaming partitioner is provided access to previous stream results [Nishimura and Ugander 2013]. Restreaming graph partitioning is motivated by environments such as online services where the same, or slightly modified, graph is repeatedly streamed with regularity. Despite adhering to the same linear memory bounds as a single-pass partitioning, the presented restreaming algorithms not only provide results comparable to METIS but also are capable of partitioning in the presence of multiple constraints and in parallel without interstream communication.

3.4.3. Vertex Cuts. A vertex cut, depicted in Figure 4(d), is equivalent to partitioning a graph by edges instead of vertices. Partitioning by edges results in each edge being assigned to one machine, while vertices are capable of spanning multiple machines. Only changes to values of cut vertices are passed over the network, not changes to edges. Vertex cuts are implemented by TLAV frameworks in response to the challenges of finding well-balanced edge cuts in power-law graphs [Abou-Rjeili and Karypis 2006; Leskovec et al. 2009]. Complex network theory suggests that power-law graphs have good vertex cuts in the form of nodes with high degree [Albert et al. 2000]. A rigorous review of vertex separators is presented in Feige et al. [2008].

PowerGraph combines vertex cuts with the three-phase GAS model (Section 3.3.1) for efficient communication and balanced computation [Gonzalez et al. 2012]. For vertices that are cut and span multiple machines, one copy is randomly designated the master, and remaining copies are mirrors. During an update, all vertices first execute a gather, where all incoming edge values are combined with a commutative associative sum operation. Then the mirrors transmit the sum value over the network to the master, which executes the apply function to produce the updated vertex value. The master then sends the result back over the network to the mirrors. Finally, each vertex completes the update by scattering the result along its outgoing edges. For each update, network traffic is proportional to the number of mirrors; therefore, breaking up high-degree vertices reduces network communication and helps to balance computation.

Since its initial implementation in PowerGraph, the vertex-cut approach has been adopted by several other TLAV frameworks. GraphX is a vertex programming abstraction for the Spark processing framework [Gonzalez et al. 2014; Zaharia et al. 2010] where the adoption of vertex cuts demonstrated an eightfold decrease in the platform's communication cost. GraphBuilder [Jain et al. 2013], an open-source graph loader, supports vertex cuts and implements grid- and torus-based vertex-cut strategies that were later included in PowerGraph. PowerLyra [Chen et al. 2015] is a modification to PowerGraph that hybridizes partitioning where vertices with a degree above a user-defined threshold are cut, while vertices below the threshold are partitioned using an adaptation of the FENNEL streaming algorithm [Tsourakakis et al. 2014]. PowerLyra also incorporates unidirectional locality similar to the GRE framework (see Section 3.2.3). BiGraph is a framework developed on PowerGraph that implements partitioning algorithms for large-scale bipartite graphs [Chen et al. 2014]. LightGraph [Zhao et al. 2014] is a framework that optimizes vertex-cut partitions by using edge-direction-aware partitioning and by not sending updates to mirrors with only in-edges.

Several edge-partitioning analyses and algorithms have recently been developed. A thorough analysis comparing expected costs of vertex partitioning and edge partitioning is presented in Bourse et al. [2014]. In this study, edge partitioning is empirically demonstrated to outperform vertex partitioning, and a streaming least-marginal-cost greedy heuristic is introduced that outperforms the greedy heuristic from PowerGraph.

Centralized hypergraph partitioning, including edge partitioning, is NP-hard, and several exact algorithms have been developed [Didi Biha and Meurs 2011; Kim and Candan 2012; Hager et al. 2014; Sevim et al. 2012]. However, because of their complexity, such algorithms are too computationally expensive and not practical for large-scale graphs. Centralized heuristics have been shown to be equally impractical [Benlic and Hao 2013]. A large-scale vertex-cut approach for bipartite graphs based on hypergraph partitioning is presented in Miao et al. [2013] as part of a vertex-centric program for computing the alternating direction of multipliers optimization technique. A distributed edge partitioner was developed in Rahimian et al. [2014] that creates balanced partitions while reducing the vertex cut, based on the vertex partitioner in Rahimian et al. [2013]. Good workload balance for skewed degree distributions can also be achieved with degree-based hashing [Xie et al. 2014]. Finally, as part of a non-vertex-centric BSP graph processing framework, a distributed vertex-cut partitioner is presented in Guerrieri and Montresor [2014] that uses a market-based model where partitions use allocated funds to buy an edge.

3.4.4. Dynamic Repartitioning. While an effective partitioning equally distributes vertices among the partitions, for TLAV frameworks, the number of active vertices performing updates on a given superstep can vary drastically over the course of computation, which creates processing imbalances and increases runtime. Dynamic repartitioning was developed to maintain balance during processing by migrating vertices between workers as necessary.

Reasons for changing active vertex sets include topological mutations to the graph and algorithmic execution properties. Topological mutations may occur if the

Table II. Feature Summary for TLAV Frameworks Implementing Dynamic Repartitioning

Framework	Cause of Imbalance	Reassignment Metric	How to Locate Migrated Verts	Densification Avoidance	Coordination
GPS	Algorithm	Sent Msgs	Broadcast Vert ID	Swap Min-Set	Decentralized
Mizan	Algorithm	Sent/Recv Msgs and Run Time	Distributed Hash Table	Metric-based Swap	Decentralized
XPregel	Algorithm	Sent/Recv Msgs	Broadcast Worker ID	Repartition Largest Worker	Centralized
xDGP	Topology	Labels of Neighbors	Broadcast Worker ID	Fraction of Capacity	Decentralized
LogGP	Both	Runtime	Lookup Table	Repartition Longest-Running Workers	Centralized
Catch the Wind	Algorithm	Sent/Recv Msgs	Lookup Table	Quota	Decentralized

Active vertex set imbalance may arise from topology changes, algorithm execution, or both. A repartitioning strategy includes how to select vertices for reassignment and how reassigned vertices are later located. Strategies should also avoid densification and can be centrally or decentrally implemented. All implemented frameworks are synchronous.

framework supports dynamic or temporal graphs (see Related Work in Section 6). Topology may also change due to the algorithm, such as graph coarsening [Wang et al. 2014].

With a static topology, the execution pattern of the algorithm can also change the active vertex set. While vertex algorithms such as synchronous PageRank execute on every vertex for every superstep, other algorithms introduce dynamism. Shang and Yu [2013] classify nine vertex algorithms as either (1) always active, (2) traversal, or (3) multiphase, where the active vertex set of the latter two classifications can vary widely and unpredictably, depending on the graph. For dynamic repartitioning to prove beneficial, the associated overhead must be less than the additional costs stemming from processing imbalance.

According to Salihoglu and Widom [2013], a dynamic repartitioning strategy must directly address (1) how to select vertices to reassign, (2) how and when to move the assigned vertices, and (3) how to locate the reassigned vertices. Other properties of a strategy include whether coordination is centralized or decentralized, and how the strategy combats "densification" and enforces vertex balance. Densification is akin to the rich-get-richer phenomenon and can occur in greedy or decentralized protocols for partitioning/clustering, where one partition becomes overpopulated as the repeated destination for migrated vertices [Vaquero et al. 2013]. In response, protocols often implement constraints that prevent a partition from exceeding a certain capacity. The XPregel framework, for example, only permits the worker with the most vertices and edges to migrate vertices [Bao and Suzumura 2013].

Table II presents six TLAV frameworks that support dynamic repartitioning: GPS [Salihoglu and Widom 2013], Mizan [Khayyat et al. 2013], XPregel [Bao and Suzumura 2013], xDGP [Vaquero et al. 2013], LogGP [Xu et al. 2014], and the Catch the Wind prototype [Shang and Yu 2013]. The table includes what active vertex set imbalances are targeted by the frameworks, what metrics are used to identify vertices for reassignment, how reassigned vertices are located after migration, how densification is avoided, and whether the protocol is centralized or decentralized.

Among the six frameworks that implement dynamic repartitioning, all are synchronous, and repartitioning occurs at the end of a superstep, separate from the updates. When a vertex is selected for migration, the worker must send all associated data to the new worker, including the vertex ID, the adjacency list, and the incoming messages to be processed in the next superstep. To avoid sending all incoming messages over the network, many dynamic repartitioning frameworks implement a form of delayed migration, where the new worker is recognized as the owner of the migrated vertex but the vertex value remains on the old worker for an extra iteration in order to compute an update. With delayed migration, the incoming message queue doesn't need to be migrated, but the new worker still receives new incoming messages [Khayyat et al. 2013; Salihoglu and Widom 2013].

Though fundamentally sound, many experiments demonstrate that dynamic repartitioning is often not worth the high overhead. Results in Bao and Suzumura [2013] show that while network I/O is significantly reduced over time, overall runtime shows minor improvements. Independent tests of GPS show dynamic repartitioning to be detrimental for all cases in Lu et al. [2014], and similar results are observed for GPS and Mizan in Han et al. [2014]. However, one major shortcoming in these evaluations is the use of the PageRank algorithm for experimentation. Dynamic repartitioning is most effective for dynamic active vertex sets but with PageRank vertices are always active, so dynamic repartitioning performs predictably poorly. Asynchronous dynamic repartitioning protocols have yet to be explored for TLAV frameworks, but the added complexity and overhead for asynchrony demonstrated in Section 3.1 suggest that such an implementation is not practical.

4. IMPLEMENTATION

This section overviews implementation details of TLAV frameworks relating to the distributed environment. These details include system architecture and fault tolerance. Additionally, TLAV frameworks that employ novel techniques to process large-scale graphs on single machines are surveyed.

4.1. System Architecture

TLAV frameworks generally always employ the master–slave architecture. A master node initializes the slave workers, monitors execution, and manages coordination (and synchronization if invoked) among the workers. Generally, the master is responsible for graph loading and partitioning, but with a network filesystem available, the loading and partitioning can be performed in parallel [Salihoglu and Widom 2013]. The master also stores global values, such as aggregators [Malewicz et al. 2010]. The workers each execute a copy of the program on the local partitions and inform the master of runtime status.

One notable exception to the general master–slave architecture is XPregel [Bao and Suzumura 2013], implemented in X10 [Charles et al. 2005]. X10 implements an Asynchronous Partitioned Global Address Space (APGAS), which is a shared address space but with a local structure that enables highly productive distributed and parallel programming. With APGAS, the number of local "places" is provided at runtime, which the programmer may utilize as necessary. XPregel does implement master–slave, but in X10, the master is actually just place 0, sans hierarchy, which opens the door for alternative architectures, like recursive structures.

4.2. Multicore Support

For multicore machines, many BSP-based frameworks including Pregel [Malewicz et al. 2010] simply assign a partition to a given core, but frameworks can better utilize computational resources through multithreading. XPregel [Bao and Suzumura 2013] supports multithreading by dividing a partition into a user-defined number of subpartitions, assigning one thread to each subpartition. GraphLab [Low et al. 2012] implements multithreading and avoids deadlocks through scheduler restrictions. GPS [Salihoglu and Widom 2013] implements three types of threads: a thread for vertex computation, a thread for communication, and a thread for parsing. Cyclops [Chen et al. 2014b] implements a hierarchical BSP model [Cha and Lee 2001] with a split design to parallelize computation and messaging while exploiting locality and avoiding synchronization contention. Cyclops demonstrates that multithreading can improve runtime relative to single-threaded execution for the same framework, at the expense of added complexity.

4.3. Fault Tolerance

Distributed systems must often account for the potential failure of one or more nodes over the course of computation. When a node fails, a replacement node may become available, but all data and computation performed on the failed node is lost.

Checkpointing is a common fault tolerance implementation, where an immutable copy of the data is written to persistent storage, such as a network filesystem. Pregel implements synchronous checkpointing, where the graph is copied in between supersteps [Malewicz et al. 2010]. When a failure occurs, the system rolls back to the most recently saved point, all partitions are reloaded, and the entire system resumes processing from the checkpoint. The partition of the failed node is reloaded to a new replacement node. If messaging information is also logged, then resources can be saved by only reloading and recomputing data on the replacement node. GraphLab [Low et al. 2012] implements asynchronous vertex checkpointing, based on Chandy-Lamport [Chandy and Lamport 1985] snapshots, which need not halt the entire program and can result in slightly faster overall execution than synchronous checkpointing, minding certain program constraints.

GraphX is a graph-processing library for Apache Spark, which is developed based on the Resilient Distributed Dataset (RDD) abstraction [Gonzalez et al. 2014]. RDDs are immutable, partitioned collections created through data-parallel operators, like map or reduce. RDDs are either stored externally or generated in-memory from operations on other RDDs. Spark maintains the lineage of operations on an RDD, so upon any node failure, the RDD can be automatically recovered. GraphX leverages the RDDs of Spark to create a graph abstraction and Pregel interface.

The Imitator [Wang et al. 2014] framework implements fault tolerance based on vertex replicas or ghosts/mirrors used in shared memory (see Section!3.2.2). The use of replicas for fault tolerance is founded in the observation that the hash partitioning of many real-world directed graphs results in the replication of over 99% of vertices [Wang et al. 2014]. By replicating *every* vertex, a full copy of the graph can reside in distributed memory, enabling faster recovery times at the expense of relatively little additional memory consumption and network messaging [Wang et al. 2014]. The efficiency of Imitator is tied to the effectiveness of the partitioning (see Section 3.4). Imitator outperforms checkpointing for large graphs distributed over several nodes, when only one replica per vertex is required. State-of-the-art partitioning methods like METIS, or a smaller number of partitions (Imitator experiments were run on 50 nodes), would likely lead to increased overhead for Imitator. Also, the number of replicas is tied to the degree of fault tolerance. To support the failure of k machines, k replicas are required, increasing overhead for each additional failure supported.

A partition-based checkpoint method for fault tolerance is presented in Shen et al. [2014]. During execution, a recovery executor node collects runtime statistics and, upon failure, uses heuristics to redistribute the partitions. Checkpointed partitions of the failed nodes can be reassigned among both new and old nodes, parallelizing recovery. Partitions on healthy nodes can also be reassigned for load balancing.

Table III. Single-Machine Frameworks

Framework	Storage Medium	Data Layout	
GraphChi	Disk/SSD	Parallel Sliding Window	[Kyrola et al. 2012]
X-Stream	Disk/SSD	Streaming Partitions	[Roy et al. 2013]
FlashGraph	SSD Array	Semiexternal Memory with Page Cache	[Zheng et al. 2015]
PathGraph	Disk/SSD	Compressed DFS Traversal Trees	[Yuan et al. 2014]

4.4. Single-Machine Architectures

Like MapReduce, TLAV frameworks are advantageous because they are highly scalable while providing a simple programming interface, abstracting away the lower-level details of distributed computing. However, such environments also stipulate the availability of elaborate infrastructure, cluster management, and performance tuning, which may not be available to all users.

Single-machine systems are easier to manage and program, but commodity machines do not have the memory capacity to process large-scale graphs in-memory. This section overviews single-machine TLAV frameworks that employ *novel* methods to process large-scale graphs. The main features of the four single-machine frameworks in this section are presented in Table III.

Processing large-scale graphs on a single machine requires either substantial amounts of memory or storing part of the graph out-of-memory, in which case performance is dictated by how efficiently the graph can be fetched from storage. In Shun and Blelloch [2013], it's argued that high-end servers, offering 100GB to 1TB of memory or more, have enough capacity for many real and synthetic graphs reported in the literature. Such machines would be capable of storing large graphs and executing relatively simple graph algorithms, though more complex algorithms would likely exhaust resources.

The recommendation service at Twitter [Gupta et al. 2013], which implements a single-machine graph-processing system with 144GB of RAM, finds that in practice, one edge occupies roughly 5 bytes of RAM on average. Compression techniques are further explored for large memory servers in Shun et al. [2015]. Yet, graphs of scale are not practical on lower-end machines containing around 8 to 16GB of memory [Kyrola et al. 2012]. Accordingly, single-machine frameworks have been developed that implement the vertex-centric programming model and process a graph in parts. Central to many single-machine TLAV frameworks are novel data layouts that efficiently read and write graph data to/from external storage. One common representation is the compressed sparse row format, which organizes graph data as outgoing edge adjacency sets, allowing for the fast lookup of outgoing edge, and has been implemented in many state-of-the-art shared-memory graph processors [Pearce et al. 2010; Hong et al. 2011], including Galois [Nguyen et al. 2013].

GraphChi. The seminal single-machine TLAV framework is GraphChi [Kyrola et al. 2012], which was explicitly developed for large-scale graph processing on a commodity desktop. GraphChi enables large-scale graph processing by implementing the Parallel Sliding Window (PSW) method, a graph data layout previously utilized for efficient PageRank and sparse-matrix dense-vector multiplication [Chen et al. 2002; Bender et al. 2007]. PSW partitions vertices into disjoint sets, associating with each interval a shard containing all of the interval's incoming edges, sorted by source vertex. Intervals are selected to form balanced shards, and the number of intervals is chosen so any interval can fit completely in memory. A sliding window is maintained over every interval, so when vertices from one shard are updated from in-edges, the results can be sequentially written to out-edges found in sorted order in the window on other shards. GraphChi may not be faster than most distributed frameworks but often reaches convergence within an order of magnitude of the performance of distributed frameworks [Kyrola et al. 2012], which is reasonable for a desktop with an order of magnitude less RAM. The GraphChi framework was later extended to a general graph management system for a single machine called GraphChi-DB [Kyrola and Guestrin 2014].

Storage concepts for single-machine graph processing are further explored in Yoneki and Roy [2013] through two directions. The first project investigates reducing random accesses in SSDs through prefetching in a project called RASP that later evolved into PrefEdge [Nilakant et al. 2014]. The second project is X-Stream [Roy et al. 2013], an edge-centric single-machine graph-processing framework that exploits the tradeoff between random memory access and sequential access from streaming data.

X-Stream. Streaming data from any storage medium provides much greater bandwidth than random access. Experiments on the X-Stream testbed, for example, demonstrate that streaming data from disk is 500 times faster than random access [Roy et al. 2013]. X-Stream combines a novel data layout, where an index is built over a storage-based edge list with an edge-centric Scatter-Gather programming model that includes a shuffle phase. Data is read from, and updates are written to, streaming edge data. Though the framework is edge centric, a user-defined update function is executed on the destination vertex of an edge. X-Stream reports that it can process a 64 billion edge graph on a single machine with a pair of 3TB magnetic disks attached [Malicevic et al. 2014].

FlashGraph. While GraphChi and X-Stream are designed for general external storage, the FlashGraph framework is developed for graphs stored on any fast I/O device, such as an array of SSDs. FlashGraph is deployed on top of the set-associative file system (SAFS) [Zheng et al. 2015], which includes a scalable lightweight page cache, and implements a custom asynchronous user-task I/O interface that reduces overhead for asynchronous I/O. FlashGraph employs asynchronous message-passing and vertex-centric programming with the semiexternal memory (SEM) model [Pearce et al. 2010], where vertices and algorithmic state reside in RAM but edges are stored externally. In experiments comparing GraphChi and X-Stream, FlashGraph outperformed both by orders of magnitude even when the data for GraphChi and X-Stream was placed into RAM disk [Zheng et al. 2015].

PathGraph. In addition to the path-centric programming model, further discussed in Section 5, PathGraph also implements a path-centric compact storage system that improves compactness and locality [Yuan et al. 2014]. Because most iterative graph algorithms involve path traversal, PathGraph stores edge-traversal trees in depth-first search order. Both the forward and reverse edge trees are each stored in a chunk storage structure that compresses data structure information including the adjacency set, vertex IDs, and the indexing of the chunk. The efficient computational model and storage structure of PathGraph resulted in improved graph loading time, lower memory footprint, and faster runtime for certain algorithms when compared to GraphChi and X-Stream.

5. ALTERNATIVE GRAPH GRANULARITY

The strengths of the vertex-centric programming model are also its weaknesses. Whereas vertex programs may be relatively simpler to reason about since only local data is available, the algorithms are less expressive than conventional centralized algorithms. While TLAV frameworks exhibit better scalability, execution can be slow because of high overhead from synchronization and message traffic that takes magnitudes longer compared to computation. Several frameworks strive for the best of

Table IV. Frameworks of Alternative Scope

Framework	Programming Model	Sequential Algorithms	Vertex Messaging	Distributed	
Giraph++	Subgraph	Y	Y	Y	[Tian et al. 2013]
Blogel	Subgraph	Y	Y	Y	[Yan et al. 2014a]
GoFFish	Subgraph	Y	N	Y	[Simmhan et al. 2014]
GraphHP	Subgraph	N	Y	Y	[Chen et al. 2014a]
P++	Subgraph	N	Y	N	[Zhou et al. 2014]
GRACE (block)	Subgraph	N	Y	N	[Xie et al. 2013]
PathGraph	Path	N	Y	Y	[Yuan et al. 2014]
Ligra	Vertex Subset	Y	Y	N	[Shun and Blelloch 2013]
Polymer	Vertex Subset	Y	Y	N	[Zhang et al. 2015]
Galois	User-Defined Set	Y	Y	N	[Nguyen et al. 2013]

both worlds by adopting a scope that is greater than a vertex but less than the graph, summarized in Table IV.

5.1. Subgraph-Centric Frameworks

Considering the challenges addressed by TLAV frameworks, taking a subgraph-centric approach is sensible. Conventional graph algorithms require the entire graph in memory, which is not possible with graphs of scale. A subgraph, though, can be partitioned into a size small enough to fit into memory (considering computation) while the connections between subgraphs would be no more, and likely much less, than the total number of edges. The system would better utilize processing while retaining scalability.

The subgraph-centric programming model is implemented in varying degrees by several frameworks. The Giraph++ [Tian et al. 2013], Blogel [Yan et al. 2014a], and GoFFish [Simmhan et al. 2014] frameworks provide a subgraph-centric interface for programming sequential algorithms. Both Giraph++ and Blogel provide a subgraph-centric interface in addition to a vertex-centric interface. The results of the sequential programs can then be shared either through vertex programs on boundary nodes or, in the case of Blogel, directly between subgraphs. GoFFish exclusively offers a subgraph-centric interface and implements messaging between subgraphs and also from subgraphs to specific vertices, the latter being used for traversal algorithms. By allowing subgraphs to directly message vertices, any vertex-centric algorithm can be implemented by a subgraph-centric framework, maintaining scalability while enabling significant performance improvement. Collectively, subgraph-centric frameworks dramatically outperform TLAV frameworks, often by *orders of magnitude* in terms of computing time, number of messages, and total supersteps [Tian et al. 2013; Yan et al. 2014a].

The GraphHP [Chen et al. 2014a] and P++ [Zhou et al. 2014] frameworks do not implement an interface for sequential programs but do differentiate between interpartition nodes to improve performance. In these two frameworks, supersteps are split into two phases: in the first phase, messages are exchanged between vertices on partition boundaries, and in the second phase, vertices within a partition repeatedly execute the vertex program to completion, exchanging messages in memory. This method reduces communication and improves performance; however, iteratively executing intraworker vertex programs is less efficient than executing a sequential algorithm. Message-passing algorithms are typically more scalable than sequential graph algorithms, but P++ is not distributed, nor is Block-based GRACE [Xie et al. 2013], an extension of Wang et al. [2013], although the latter demonstrates that executing vertex updates on a subgraph block basis improves locality and cache hits while reducing memory access time, which is a bottleneck for computationally light algorithms like PageRank.

TLAV frameworks illustrate the principal ideas for scalable graph processing, but for the best performance, users may consider subgraph-centric frameworks. Subgraph frameworks leverage principles of TLAV frameworks to execute sequential graph algorithms in a distributed environment. The Giraph++, Blogel, and GoFFish frameworks reduce the scope of sequential graph algorithms for the subgraph to fit in memory while utilizing vertex or subgraph messaging to maintain scalability. Together, the vertex-centric and subgraph-centric programming model, compared to sequential graph algorithms, demonstrate how scalability varies inversely with scope.

5.2. Other Scopes: Paths and Sets

While subgraph-centric frameworks illustrate the scope/scalability tradeoff, several other frameworks adopt alternative computational scopes that demonstrate additional benefits.

A more specific type of subgraph, a traversal tree, is used for the programming model in PathGraph [Yuan et al. 2014]. Traversals are a fundamental component of many graph algorithms, including PageRank and Bellman-Ford shortest path. PathGraph first partitions the graph into paths, with each partition represented as two trees, a forward and reverse edge traversal. Then, for the path-centric computational model, path-centric scatter and path-centric gather functions are available to the user to define an algorithm that traverses each tree. The user also defines a vertex update function, which is executed by the path-centric functions during the traversal. Like block-based GRACE, the path-centric model utilizes locality to improve performance through reduced memory usage and efficient caching. PathGraph also implements a path-centric storage model that enables the framework to process billion-node graphs on a single machine (see Section 4.4) [Yuan et al. 2014].

Graph-processing frameworks designed for single machines can implement interfaces of unique granularity. A vertex subset interface is implemented in Ligra [Shun and Blelloch 2013]. Ligra argues that high-end servers provide enough memory for large-scale graphs, and thus implements a vertex-centric programming interface while retaining a global view of the graph. Inspired by a hybrid breadth-first search (BFS) algorithm [Beamer et al. 2013], Ligra dynamically switches between sparse and dense representations of edge sets depending on the size of the vertex subset, which impacts whether push or pull operations are performed with the vertex subset. Polymer [Zhang et al. 2015] adopts a similar interface as Ligra, but with several NUMA-aware optimizations. Galois [Kulkarni et al. 2009] is a shared-memory framework that executes user-defined set operators while exploiting amorphous data parallelism [Pingali et al. 2011]. Galois can be implemented in a variety of programming interfaces, including the vertex-centric paradigm [Nguyen et al. 2013].

5.3. Optimizations

Two optimizations have been introduced in Salihoglu and Widom [2014] for TLAV frameworks that improve performance by adopting a scope of the graph other than vertex centric. The Finishing Computation Serially (FCS) method is applicable when an algorithm with a shrinking set of active vertices converges slowly near the end of execution [Salihoglu and Widom 2014]. The FCS method is triggered when the remaining active graph can fit in the memory of a single machine; in these instances, the active portions are sent to the master and completed serially from a global, shared-memory perspective of the graph.

Similarly, the Single Pivot (SP) optimization [Salihoglu and Widom 2014], first presented in Quick et al. [2012], also temporarily adopts a global view. For algorithms that execute BFS across all vertices (e.g., the connected components algorithm), instead of executing BFS from every node, which incurs a high messaging cost, SP randomly selects one vertex from the graph and performs BFS just from that vertex. Since most graphs have one big component in addition to many small ones, the BFS from a random node can be executed until the big component is found, and then BFS from every vertex that's not in the big component can execute BFS to complete the algorithm, resulting in significantly fewer total messages. This optimization adjusts scope by randomly selecting a single vertex by utilizing a global aggregator [Malewicz et al. 2010], which also adopts a scope beyond vertex.

6. RELATED WORK

In this article, vertex-centric graph-processing systems for large-scale graphs are surveyed. In previous related work, Pregel and GraphLab have been compared [Sakr 2013], general graph-processing systems have been surveyed [Khan and Elnikety 2014; Nisar et al. 2013], and four TLAV frameworks have been empirically evaluated on four algorithms [Han et al. 2014]. A tutorial on TLAV frameworks was recently delivered at an international conference [Ajwani et al. 2015].

TLAV frameworks intersect several subjects, including graph processing, distributed computing, Big Data, and distributed algorithms. Several graph-processing frameworks have been recently developed outside of the vertex-centric programming model. PEGASUS combines the BSP model with generalized matrix-vector multiplication (GIM-V) [Kang et al. 2011], while TurboGraph introduces the pin-and-slide model to perform GIM-V on a single machine [Han et al. 2013]. Combinatorial BLAS [Bulu c̃c and Gilbert 2011] and the Parallel Boost Graph Library [Gregor and Lumsdaine 2005] are software libraries for high-performing parallel computation of sequential programs. Piccolo performs distributed graph computation using distributed tables [Power and Li 2010].

Graph databases, such as Neo4j [Webber 2012], HyperGraphDB [Iordanov 2010], and GBASE [Kang et al. 2011], are decidedly different from TLAV frameworks. Both treat vertices as first-class citizens, and both face related problems like partitioning, but the key distinction is that databases focus on transactional processing while TLAV frameworks focus on batch processing [Chen et al. 2012]. Databases offer local or online queries, such as one-hop neighbors, whereas TLAV systems iteratively process the entire graph offline in batch. Some more general graph management systems, like Trinity [Shao et al. 2013] and Grace [Prabhakaran et al. 2012], offer suites of features that include both vertex-centric processing and queries. Sensibly, a graph-processing engine may be developed on top of a graph database. However, the two should not be confused, and performance is incomparable.

A closely related Big Data framework is MapReduce [Dean and Ghemawat 2008; Polato et al. 2014]. MapReduce is a different programming model from TLAV frameworks but similarly enables large-scale computation and, when implemented, abstracts away the details of distributed programming. The programming model is effective for many types of computation but addresses neither iterative processing nor graph processing [Polato et al. 2014; Malewicz et al. 2010]. Iterative computation is not natively supported, as the programming model performs only a single pass over the data with no loop awareness. Moreover, I/O is read/written to/from a distributed filesystem (e.g., HDFS), rendering iterative computation inefficient [Polato et al. 2014]. Nonetheless, several frameworks have extended MapReduce to support iterative computation [Ekanayake et al. 2010; Bu et al. 2012; Zhang et al. 2012b], but such frameworks are still agnostic to the challenges of graph processing. Graph computation with MapReduce has been explored [Lin and Schatz 2010] but is generally acknowledged to be lacking [Cohen 2009; Malewicz et al. 2010]. A comparison of MapReduce and BSP is provided in Kajdanowicz et al. [2014]. Still, some argue that MapReduce should remain the sole “hammer” for Big Data analytics because of the widespread adoption throughout industry [Lin 2013].

Similarly, in response to TLAV shortcomings, such as poor out-of-core support and lengthy loading times, some frameworks rework preexisting graph database technologies to provide a vertex-centric interface [Fan et al. 2015]. However, many of these projects lose sight of the main problems addressed by the vertex-centric processing. TLAV frameworks are ultimately Big Data solutions, designed large graphs to be leveraged against the memory and processing power of several machines, not single machines. Moreover, TLAV frameworks iteratively process the entire graph and do not provide graph queries like one-hop or two-hop neighbors. TLAV frameworks are not a universal solution for graph analytics but rather provide an approach for scalable, iterative graph processing.

Temporal graph processing is beyond the scope of this survey, though a small number of TLAV frameworks have been developed for temporal analysis [Cheng et al. 2012; Hant et al. 2014]. These frameworks compute temporal properties offline in batch through graph snapshots, necessitating multiple framework components, including a front-end ingress component, an analytics engine, and a storage component such as a graph database. Temporal graph layout optimizations were introduced in Chronos [Hant et al. 2014]. These frameworks illustrate how advanced graph analytics systems utilize the strengths of different graph technologies for different components, for example, graph databases for storage and online queries, and vertex-centric computation for batch analytics. Dynamic graph algorithms and general analytics systems have also been surveyed [Aggarwal and Subbian 2014; Vaquero et al. 2014]. Dynamic graphs are supported by many frameworks including Pregel, but the topic was omitted from this survey due to widely varying support by the frameworks and broad scope of the topic.

While coined “vertex centric” relative to conventional graph-processing approaches, the algorithms executed by TLAV frameworks are more formally known as distributed algorithms. Distributed algorithms are a mature field of study [Lynch 1996], and further examples beyond Figure 3 may be found within the referenced frameworks. Some works have explored distributed algorithms within the context of TLAV frameworks [Yan et al. 2014b], but researchers and practitioners should be aware that TLAV frameworks execute distributed algorithms [Lynch 1996], which come from a field with a considerable body of work, including theory and analysis. The theoretical limits of what can be computed with vertex-centric frameworks, specifically with the synchronous, message-passing LOCAL model, have been studied [Kuhn et al. 2010].

This article surveys and compares the various components of TLAV frameworks, which are a platform for executing vertex-centric algorithms. Like MapReduce, these frameworks provide an interface for a user-defined function while abstracting away the lower-level details of cluster computing. Changing the components of the framework will impact system performance and runtime characteristics but will generally not impact the design or result of the algorithm.[5]

7. CONCLUSIONS

TLAV frameworks have been designed in response to the challenges of processing large graphs. Primary challenges include the unstructured nature of graphs, where an edge may span any two vertices, so the entire graph must be randomly accessible for conventional processing. TLAV frameworks are also developed for ease of use, providing a simple vertex-centric interface while abstracting away the lower-level details of cluster

[5]An exception to this rule is synchronous versus asynchronous execution of some algorithms, such as graph coloring in Section 3.1.

computing. MapReduce similarly enables highly scalable computing but is ill-suited for iterative graph processing.

By adopting a vertex-centric programming model, the scope of computation is dramatically reduced. To perform an update, each vertex only needs data from immediate neighbors. Data residing on a separate machine can be acquired directly between workers, avoiding the bottleneck of central coordination and enabling excellent scalability. The four pillars of the vertex-centric programming model, (1) timing, (2) communication, (3) the execution model, and (4) partitioning, were presented and surveyed in the context of distributed graph-processing frameworks. However, vertex-centric algorithms, colloquially known as distributed algorithms, have an established history and are still actively researched [Lynch 1996; Kuhn et al. 2010].

Several related frameworks were explored that similarly adopt a computational scope of the graph at varying granularities. These frameworks of alternative scope are like a Goldilocks solution to graph processing. Centralized algorithms with the entire graph in scope require too much memory and vertex-centric algorithms can scale but are less expressive and require many relatively slow messages, whereas subgraph-centric algorithms can utilize the two resources just right. A significant contribution of TLAV frameworks is exposing how, for graphs, reducing the scope of a program increases scalability.

Of course, expressing a particular algorithm as subgraph centric is not trivial. The future of practical large-scale distributed graph processing may be related to finding algorithms that process a graph as independent subgraphs, such as divide and conquer, or algorithms that can process graphs at multiple, or even dynamic, scopes [Wang et al. 2014]. The performance of the subgraph-centric processing is also closely tied to the effectiveness of large-scale graph partitioning, including streaming and distributed partitioning techniques.

TLAV frameworks are a tool for graph processing at scale. Not all graphs are large enough to necessitate distributed processing, and not all graph problems need the whole graph to be computed iteratively. Moreover, there is often more than one way to solve a problem, but these frameworks are simple to program, easy to distribute, and not a bad choice for the right type of problem. Subgraph-centric frameworks take vertex-centric frameworks a step further for performance. Datasets will continue to grow dramatically into the new age of Big Data, and the design of processing systems should begin asking if they can scale out infinitely. TLAV frameworks illustrate how conventional centralized systems will fail in the Big Data ecosystem, and how decentralized platforms must be embraced.

REFERENCES

Amine Abou-Rjeili and George Karypis. 2006. Multilevel algorithms for partitioning power-law graphs. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing (IPDPS'06)*. IEEE Computer Society, Washington, DC, 124. http://dl.acm.org/citation.cfm?id=1898953.1899055.

Charu Aggarwal and Karthik Subbian. 2014. Evolutionary network analysis: A survey. *ACM Comput. Surv.* 47, 1, Article 10 (May 2014), 36 pages. DOI: http://dx.doi.org/10.1145/2601412

Amr Ahmed, Nino Shervashidze, Shravan Narayanamurthy, Vanja Josifovski, and Alexander J. Smola. 2013. Distributed large-scale natural graph factorization. In *Proceedings of the 22nd International Conference on World Wide Web (WWW'13)*. International World Wide Web Conferences Steering Committee, Geneva, Switzerland, 37–48. http://dl.acm.org/citation.cfm?id=2488388.2488393.

Deepak Ajwani, Marcel Karnstedt, and Alessandra Sala. 2015. Processing large graphs: Representations, storage, systems and algorithms. In *Proceedings of the 24th International Conference on World Wide Web Companion (WWW'15 Companion)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 1545–1545. DOI: http://dx.doi.org/10.1145/2740908.2741990

Réka Albert, Hawoong Jeong, and Albert-László Barabási. 2000. Error and attack tolerance of complex networks. *Nature* 406, 6794 (2000), 378–382. DOI: http://dx.doi.org/10.1038/35019019

Konstantin Andreev and Harald Racke. 2006. Balanced graph partitioning. *Theory Comput. Syst.* 39, 6, 929–939.

Ching Avery. 2011. Giraph: Large-scale graph processing infrastructure on Hadoop. In *Proceedings of Hadoop Summit*. Santa Clara, CA.

Nguyen Thien Bao and Toyotaro Suzumura. 2013. Towards highly scalable pregel-based graph processing platform with x10. In *Proceedings of the 22nd International Conference on World Wide Web Companion (WWW'13 Companion)*. International World Wide Web Conferences Steering Committee, Geneva, Switzerland, 501–508. http://dl.acm.org/citation.cfm?id=2487788.2487984.

Scott Beamer, Krste Asanović, and David Patterson. 2013. Direction-optimizing breadth-first search. *Sci. Program.* 21, 3–4 (July 2013), 137–148. http://dl.acm.org/citation.cfm?id=2590251.2590258.

Richard Bellman. 1958. On a routing problem. *Quart. Appl. Math.* 16 (1958), 87–90.

Michael A. Bender, Gerth Stølting Brodal, Rolf Fagerberg, Riko Jacob, and Elias Vicari. 2007. Optimal sparse matrix dense vector multiplication in the I/O-model. In *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'07)*. ACM, New York, NY, 61–70. DOI:http://dx.doi.org/10.1145/1248377.1248391

Una Benlic and Jin-Kao Hao. 2013. Breakout local search for the vertex separator problem. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI'13)*. AAAI Press, 461–467. http://dl.acm.org/citation.cfm?id=2540128.2540196.

Dimitri P. Bertsekas, Francesca Guerriero, and Roberto Musmanno. 1996. Parallel asynchronous label-correcting methods for shortest paths. *J. Optim. Theory Appl.* 88, 2 (Feb. 1996), 297–320. DOI:http://dx.doi.org/10.1007/BF02192173

Dimitri P. Bertsekas and John N. Tsitsiklis. 1989. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Upper Saddle River, NJ.

Florian Bourse, Marc Lelarge, and Milan Vojnovic. 2014. Balanced graph edge partition. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'14)*. ACM, New York, NY, 1456–1465. DOI:http://dx.doi.org/10.1145/2623330.2623660

Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.* 30, 1–7 (April 1998), 107–117. DOI:http://dx.doi.org/10.1016/S0169-7552(98)00110-X

Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. 2012. The haloop approach to large-scale iterative data analysis. *VLDB J.* 21, 2 (April 2012), 169–190. DOI:http://dx.doi.org/10.1007/s00778-012-0269-7

Aydín Buluc̃c and John R Gilbert. 2011. The combinatorial BLAS: Design, implementation, and applications. *Int. J. High Perform. Comput. Appl.* 25, 4 (Nov. 2011), 496–509. DOI:http://dx.doi.org/10.1177/1094342011403516

Hojung Cha and Dongho Lee. 2001. H-BSP: A hierarchical BSP computation model. *J. Supercomput.* 18, 2 (Feb. 2001), 179–200. DOI:http://dx.doi.org/10.1023/A:1008113017444

Kanianthra Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.* 3, 1 (Feb. 1985), 63–75. DOI:http://dx.doi.org/10.1145/214451.214456

Kanianthra Mani Chandy and Jayadev Misra. 1984. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.* 6, 4 (Oct. 1984), 632–646. DOI:http://dx.doi.org/10.1145/1780.1804

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.* 40, 10 (Oct. 2005), 519–538. DOI:http://dx.doi.org/10.1145/1103845.1094852

Qun Chen, Song Bai, Zhanhuai Li, Zhiying Gou, Bo Suo, and Wei Pan. 2014a. GraphHP: A hybrid platform for iterative graph processing. Retrieved July 17, 2014, from http://wowbigdata.net.cn/paper/GraphHP%EF%BC%9AA%20Hybrid%20Platform%20for%20Iterative%20Graph%20Processing.pdf.

Rong Chen, Xin Ding, Peng Wang, Haibo Chen, Binyu Zang, and Haibing Guan. 2014b. Computation and communication efficient graph processing with distributed immutable view. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC'14)*. ACM, New York, NY, 215–226. DOI:http://dx.doi.org/10.1145/2600212.2600233

Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)*. ACM, New York, NY, Article 1, 15 pages. DOI:http://dx.doi.org/10.1145/2741948.2741970

Rong Chen, Jiaxin Shi, Binyu Zang, and Haibing Guan. 2014. Bipartite-oriented distributed graph partitioning for big learning. In *Proceedings of 5th Asia-Pacific Workshop on Systems (APSys'14)*. ACM, Article 14, 7 pages. DOI:http://dx.doi.org/10.1145/2637166.2637236

Rishan Chen, Mao Yang, Xuetian Weng, Byron Choi, Bingsheng He, and Xiaoming Li. 2012. Improving large graph processing on partitioned graphs in the cloud. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC'12)*. ACM, New York, NY, Article 3, 13 pages. DOI:http://dx.doi.org/10.1145/2391229.2391232

Yen-Yu Chen, Qingqing Gan, and Torsten Suel. 2002. I/O-efficient techniques for computing pagerank. In *Proceedings of the 11th International Conference on Information and Knowledge Management (CIKM'02)*. ACM, New York, NY, 549–557. DOI:http://dx.doi.org/10.1145/584792.584882

Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*. ACM, New York, NY, 85–98. DOI:http://dx.doi.org/10.1145/2168836.2168846

Sun Chung and Anne Condon. 1996. Parallel implementation of Bouvka's minimum spanning tree algorithm. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS'06)*. IEEE Computer Society, Washington, DC, 302–308. DOI:http://dx.doi.org/10.1109/IPPS.1996.508073

Jonathan Cohen. 2009. Graph twiddling in a MapReduce world. *Comput. Sci. Eng.* 11, 4 (July 2009), 29–41. DOI:http://dx.doi.org/10.1109/MCSE.2009.120

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. DOI:http://dx.doi.org/10.1145/1327452.1327492

Mohamed Didi Biha and Marie-Jean Meurs. 2011. An exact algorithm for solving the vertex separator problem. *J. Global Optim.* 49, 3 (March 2011), 425–434. DOI:http://dx.doi.org/10.1007/s10898-010-9568-y

Edsger Wybe Dijkstra. 1959. A note on two problems in connection with graphs. *Numer. Math.* 1, 1 (1959), 269–271.

Edsger Wybe Dijkstra. 1971. Hierarchical ordering of sequential processes. *Acta Inf.* 1, 2 (June 1971), 115–138. DOI:http://dx.doi.org/10.1007/BF00289519

Nicholas Edmonds. 2013. *Active Messages as a Spanning Model for Parallel Graph Computation*. Ph.D. Dissertation. Indiana University.

Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. 2010. Twister: A runtime for iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC'10)*. ACM, New York, NY, 810–818. DOI:http://dx.doi.org/10.1145/1851476.1851593

Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. 2015. The case against specialized graph analytics engines. In *Online Proceedings of the 7th Biennial Conference on Innovative Data Systems Research (CIDR'15),* Asilomar, CA, January 4–7, 2015. http://www.cidrdb.org/cidr2015/Papers/CIDR15_Paper20.pdf.

Uriel Feige, MohammadTaghi Hajiaghayi, and James R. Lee. 2008. Improved approximation algorithms for minimum weight vertex separators. *SIAM J. Comput.* 38, 2 (May 2008), 629–657. DOI:http://dx.doi.org/10.1137/05064299X

Linton C. Freeman. 1977. A set of measures of centrality based on betweenness. *Sociometry* 40, 1, 35–41.

Joachim Gehweiler and Henning Meyerhenke. 2010. A distributed diffusive heuristic for clustering a virtual P2P supercomputer. In *Proceedings of the 2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and PhD Forum (IPDPSW'10)*. 1–8. DOI:http://dx.doi.org/10.1109/IPDPSW.2010.5470922

Joseph Gonzalez, Yucheng Low, Arthur Gretton, and Carlos Guestrin. 2011. Parallel gibbs sampling: From colored fields to thin junction trees. In *International Conference on Artificial Intelligence and Statistics*. 324–332.

Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, 17–30. http://dl.acm.org/citation.cfm?id=2387880.2387883.

Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, 599–613. http://dl.acm.org/citation.cfm?id=2685048.2685096

Douglas Gregor and Andrew Lumsdaine. 2005. The parallel BGL: A generic library for distributed graph computations. In *Proceedings of the Parallel Object-Oriented Scientific Computing (POOSC'14)*.

Alessio Guerrieri and Alberto Montresor. 2014. Distributed edge partitioning for graph processing. *arXiv preprint arXiv:1403.6270*. http://arxiv.org/abs/1403.6270

Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. 2013. WTF: The who to follow service at twitter. In *Proceedings of the 22nd International Conference on World Wide*

Web (WWW'13). International World Wide Web Conferences Steering Committee, Geneva, Switzerland, 505–514. http://dl.acm.org/citation.cfm?id=2488388.2488433

William W. Hager, James T. Hungerford, and Ilya Safro. 2014. A multilevel bilinear programming algorithm for the vertex separator problem. *arXiv preprint arXiv:1410.4885*. http://arxiv.org/abs/1410.4885.

Minyang Han, Khuzaima Daudjee, Khaled Ammar, M. Tamer Ozsu, Xingfang Wang, and Tianqi Jin. 2014. An experimental comparison of pregel-like graph processing systems. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1047–1058.

Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. 2013. TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'13)*. ACM, New York, NY, 77–85. DOI:http://dx.doi.org/10.1145/2487575.2487581

Wentao Hant, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. 2014. Chronos: A graph engine for temporal graph analysis. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys'14)*. ACM, New York, NY, Article 1, 14 pages. DOI:http://dx.doi.org/10.1145/2592798.2592799

Harshvardhan, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. 2014. KLA: A new algorithmic paradigm for parallel graph computations. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT'14)*. ACM, New York, NY, 27–38. DOI:http://dx.doi.org/10.1145/2628071.2628091

Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. 2011. Efficient parallel graph exploration on multi-core CPU and GPU. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*. IEEE Computer Society, Washington, DC, 78–88. DOI:http://dx.doi.org/10.1109/PACT.2011.14

Imranul Hoque and Indranil Gupta. LFGraph: Simple and fast distributed graph analytics. In *Proceedings of the ACM Symposium on Timely Results in Operating Systems*.

Borislav Iordanov. 2010. HyperGraphDB: A generalized graph database. In *Proceedings of the 2010 International Conference on Web-Age Information Management*. Springer-Verlag, Berlin, 25–36. http://dl.acm.org/citation.cfm?id=1927585.1927589

Nilesh Jain, Guangdeng Liao, and Theodore L. Willke. 2013. GraphBuilder: Scalable graph ETL framework. In *1st International Workshop on Graph Data Management Experiences and Systems (GRADES'13)*. ACM, New York, NY, Article 4, 6 pages. DOI:http://dx.doi.org/10.1145/2484425.2484429

Tomasz Kajdanowicz, Przemyslaw Kazienko, and Wojciech Indyk. 2014. Parallel processing of large graphs. *Future Gener. Comput. Syst.* 32 (March 2014), 324–337. DOI:http://dx.doi.org/10.1016/j.future.2013.08.007

U Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. 2011. GBASE: A scalable and general graph management system. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'11)*. ACM, New York, NY, 1091–1099. DOI:http://dx.doi.org/10.1145/2020408.2020580

U Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. 2009. PEGASUS: A peta-scale graph mining system implementation and observations. In *Proceedings of the 2009 9th IEEE International Conference on Data Mining (ICDM'09)*. IEEE Computer Society, Washington, DC, 229–238. DOI:http://dx.doi.org/10.1109/ICDM.2009.14

George Karypis and Vipin Kumar. 1995. Multilevel graph partitioning schemes. In *Proceedings of the International Conference on Parallel Processing (ICPP'95)*. 113–122.

George Karypis and Vipin Kumar. 1996. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing (Supercomputing'96)*. IEEE Computer Society, Washington, DC, Article 35. DOI:http://dx.doi.org/10.1145/369028.369103

Brian W. Kernighan and Shen Lin. 1970. An efficient heuristic procedure for partitioning graphs. *Bell Syst. Tech. J.* 49, 2 (1970), 291–307. DOI:http://dx.doi.org/10.1002/j.1538-7305.1970.tb01770.x

Arijit Khan and Sameh Elnikety. 2014. Systems for big-graphs. *Proc. VLDB Endow.* 7, 13 (Aug. 2014), 1709–1710. http://dl.acm.org/citation.cfm?id=2733004.2733067

Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. 2013. Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)*. ACM, New York, NY, 169–182. DOI:http://dx.doi.org/10.1145/2465351.2465369

Gang-Hoon Kim, Silvana Trimi, and Ji-Hyong Chung. 2014. Big-data applications in the government sector. *Commun. ACM* 57, 3 (March 2014), 78–85. DOI:http://dx.doi.org/10.1145/2500873

Mijung Kim and K. Selçuk Candan. 2012. SBV-Cut: Vertex-cut based graph partitioning using structural balance vertices. *Data Knowl. Eng.* 72 (Feb. 2012), 285–303. DOI:http://dx.doi.org/10.1016/j.datak.2011.11.004

Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. 2010. Local computation: Lower and upper bounds. *arXiv preprint arXiv:1011.5470*.

Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2009. Optimistic parallelism requires abstractions. *Commun. ACM* 52, 9 (Sept. 2009), 89–97. DOI:http://dx.doi.org/10.1145/1562164.1562188

Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, 31–46. http://dl.acm.org/citation.cfm?id=2387880.2387884

Aapo Kyrola and Carlos Guestrin. 2014. GraphChi-DB: Simple design for a scalable graph database system–on just a PC. *arXiv preprint arXiv:1403.0701*. http://arxiv.org/abs/1403.0701

Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. 2009. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Math.* 6, 1 (2009), 29–123.

Jimmy Lin. 2013. Mapreduce is good enough? If all you have is a hammer, throw away everything that's not a nail! *Big Data* 1, 1 (2013), 28–37.

Jimmy Lin and Michael Schatz. 2010. Design patterns for efficient graph algorithms in MapReduce. In *Proceedings of the 8th Workshop on Mining and Learning with Graphs (MLG'10)*. ACM, New York, NY, 78–85. DOI:http://dx.doi.org/10.1145/1830252.1830263

Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.* 5, 8 (April 2012), 716–727. DOI:http://dx.doi.org/10.14778/2212351.2212354

Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2010. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1006.4990*.

Honghui Lu, Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. 1995. Message passing versus distributed shared memory on networks of workstations. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (Supercomputing'95)*. ACM, New York, NY, Article 37. DOI:http://dx.doi.org/10.1145/224170.224285

Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. 2014. Large-scale distributed graph computing systems: An experimental evaluation. *Proc. VLDB Endow.* 8 (2014), 3.

Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. 2007. Challenges in parallel graph processing. *Parallel Process. Lett.* 17, 01 (2007), 5–20. DOI:http://dx.doi.org/10.1142/S0129626407002843

Nancy A. Lynch. 1996. *Distributed Algorithms*. Morgan Kaufmann.

Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*. ACM, New York, NY, 135–146. DOI:http://dx.doi.org/10.1145/1807167.1807184

Jasmina Malicevic, Laurent Bindschaedler, Amitabha Roy, and Willy Zwaenepoel. 2014. X-Stream. http://labos.epfl.ch/x-stream.

Urlich Meyer and Peter Sanders. 2003. Δ-stepping: A parallelizable shortest path algorithm. *J. Algorithms* 49, 1 (2003), 114–152. DOI:http://dx.doi.org/10.1016/S0196-6774(03)00076-2 1998 European Symposium on Algorithms.

Henning Meyerhenke, Peter Sanders, and Christian Schulz. 2014. Parallel graph partitioning for complex networks. *arXiv preprint arXiv:1404.4797*.

Hui Miao, Xiangyang Liu, Bert Huang, and Lise Getoor. 2013. A hypergraph-partitioned vertex programming approach for large-scale consensus optimization. In *Proceedings of the 2013 IEEE International Conference on Big Data*. 193–198.

Kameshwar Munagala and Abhiram Ranade. 1999. I/O-complexity of graph algorithms. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'99)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 687–694. http://dl.acm.org/citation.cfm?id=314500.314891.

Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, New York, NY, 456–471. DOI:http://dx.doi.org/10.1145/2517349.2522739

Karthik Nilakant, Valentin Dalibard, Amitabha Roy, and Eiko Yoneki. 2014. PrefEdge: SSD prefetcher for large-scale graph traversal. In *Proceedings of International Conference on Systems and Storage (SYSTOR'14)*. ACM, New York, NY, Article 4, 12 pages. DOI:http://dx.doi.org/10.1145/2611354.2611365

M. Usman Nisar, Arash Fard, and John A. Miller. 2013. Techniques for graph analytics on big data. In *Proceedings of the 2013 IEEE International Congress on Big Data (BIGDATACONGRESS'13)*. IEEE Computer Society, Washington, DC, 255–262. DOI:http://dx.doi.org/10.1109/BigData.Congress.2013.78

Joel Nishimura and Johan Ugander. 2013. Restreaming graph partitioning: Simple versatile algorithms for advanced balancing. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'13)*. ACM, New York, NY, 1106–1114. DOI:http://dx.doi.org/10.1145/2487575.2487696

Bill Nitzberg and Virginia Lo. 1991. Distributed shared memory: A survey of issues and algorithms. *Computer* 24, 8 (Aug. 1991), 52–60. DOI:http://dx.doi.org/10.1109/2.84877

Matthew Felice Pace. 2012. {BSP} vs MapReduce. *Procedia Comput. Sci.* 9, (2012), 246–255. DOI:http://dx.doi.org/10.1016/j.procs.2012.04.026 Proceedings of the International Conference on Computational Science, {ICCS} 2012.

Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report. Stanford InfoLab.

Roger Pearce, Maya Gokhale, and Nancy M. Amato. 2010. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*. IEEE Computer Society, Washington, DC, 1–11. DOI:http://dx.doi.org/10.1109/SC.2010.34

David Peleg. 2000. *Distributed Computing: A Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA.

Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The tao of parallelism in algorithms. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. 12–25. DOI:http://dx.doi.org/10.1145/1993498.1993501

Ivanilton Polato, Reginaldo R. Alfredo Goldman, and Fabio Kon. 2014. A comprehensive view of Hadoop research – A systematic literature review. *J. Network Comput. Appl.* 46, (2014), 1–25. DOI:http://dx.doi.org/10.1016/j.jnca.2014.07.022

Russell Power and Jinyang Li. 2010. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, Vol. 10. 1–14.

Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haridasan. 2012. Managing large graphs on multi-cores with graph awareness. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12)*. USENIX Association, Berkeley, CA, 4. http://dl.acm.org/citation.cfm?id=2342821.2342825.

Robert Preis. 1999. Linear time 1/2 -approximation algorithm for maximum weighted matching in general graphs. In *Proceedings of the 16th Annual Conference on Theoretical Aspects of Computer Science (STACS'99)*. Springer-Verlag, Berlin, 259–269. http://dl.acm.org/citation.cfm?id=1764891.1764924.

Jelica Protic, Milo Tomasevic, and Veljko Milutinovic (Eds.). 1997. *Distributed Shared Memory: Concepts and Systems*. IEEE Computer Society Press, Los Alamitos, CA.

Louise Quick, Paul Wilkinson, and David Hardcastle. 2012. Using pregel-like large scale graph processing frameworks for social network analysis. In *Proceedings of the 2012 International Conference on Advances in Social Networks Analysis and Mining (ASONAM'12)*. IEEE Computer Society, Washington, DC, 457–463. DOI:http://dx.doi.org/10.1109/ASONAM.2012.254

Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. 2007. Near linear time algorithm to detect community structures in large-scale networks. *Phys. Rev. E* 76, 3 (2007), 036106. http://dx.doi.org/10.1103/PhysRevE.76.036106.

Fatemeh Rahimian, Amir H. Payberah, Sarunas Girdzijauskas, and Seif Haridi. 2014. Distributed vertex-cut partitioning. In *Distributed Applications and Interoperable Systems*, Kostas Magoutis and Peter Pietzuch (Eds.). Springer, Berlin, 186–200. DOI:http://dx.doi.org/10.1007/978-3-662-43352-2_15

Fatemeh Rahimian, Amir H. Payberah, Sarunas Girdzijauskas, Mark Jelasity, and Seif Haridi. 2013. JA-BE-JA: A distributed algorithm for balanced graph partitioning. In *Proceedings of the 2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems (SASO'13)*. IEEE Computer Society, Washington, DC, 51–60. DOI:http://dx.doi.org/10.1109/SASO.2013.13

Lakshmish Ramaswamy, Bugra Gedik, and Ling Liu. 2005. A distributed approach to node clustering in decentralized peer-to-peer networks. *IEEE Trans. Parallel Distrib. Syst.* 16, 9 (Sept. 2005), 814–829. DOI:http://dx.doi.org/10.1109/TPDS.2005.101

Mark Redekopp, Yogesh Simmhan, and Viktor K. Prasanna. 2013. Optimizations and analysis of BSP graph processing models on public clouds. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS'13)*. IEEE Computer Society, Washington, DC, 203–214. DOI:http://dx.doi.org/10.1109/IPDPS.2013.76

Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, New York, NY, 472–488. DOI:http://dx.doi.org/10.1145/2517349.2522740

Sherif Sakr. 2013. Processing large-scale graph data: A guide to current technology. *IBM Developerworks* (June 2013), 15.

Semih Salihoglu and Jennifer Widom. 2013. GPS: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management (SSDBM'13)*. ACM, New York, NY, Article 22, 12 pages. DOI:http://dx.doi.org/10.1145/2484838.2484843

Semih Salihoglu and Jennifer Widom. 2014. *Optimizing Graph Algorithms on Pregel-Like Systems*. Technical Report. Stanford InfoLab.

Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. 2014. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD'14)*. ACM, New York, NY, 979–990. DOI:http://dx.doi.org/10.1145/2588555.2610518

Sangwon Seo, Edward J. Yoon, Jaehong Kim, Seongwook Jin, Jin-Soo Kim, and Seungryoul Maeng. 2010. HAMA: An efficient matrix computation with the MapReduce framework. In *Proceedings of the 2010 IEEE 2nd International Conference on Cloud Computing Technology and Science (CLOUDCOM'10)*. IEEE Computer Society, Washington, DC, 721–726. DOI:http://dx.doi.org/10.1109/CloudCom.2010.17

Tina Beseri Sevim, Hakan Kutucu, and Murat Ersen Berberler. 2012. New mathematical model for finding minimum vertex cut set. In *2012 IV International Conference on Problems of Cybernetics and Informatics (PCI'12)*. 1–2. DOI:http://dx.doi.org/10.1109/ICPCI.2012.6486469

Zechao Shang and Jeffrey Xu Yu. 2013. Catch the wind: Graph workload balancing on cloud. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE'13)*. IEEE Computer Society, Washington, DC, 553–564. DOI:http://dx.doi.org/10.1109/ICDE.2013.6544855

Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. ACM, New York, NY, 505–516. DOI:http://dx.doi.org/10.1145/2463676.2467799

Yanyan Shen, Gang Chen, H. V. Jagadish, Wei Lu, Beng Chin Ooi, and Bogdan Marius Tudor. 2014. Fast failure recovery in distributed graph processing systems. *Proc. VLDB Endow.* 8, 4 (Dec. 2014), 437–448. http://dl.acm.org/citation.cfm?id=2735496.2735506.

Julian Shun and Guy E. Blelloch. 2013. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'13)*. ACM, New York, NY, 135–146. DOI:http://dx.doi.org/10.1145/2442516.2442530

Julian Shun, Laxman Dhulipala, and Guy Blelloch. 2015. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *Proceedings of the IEEE Data Compression Conference (DCC'15)*.

Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, and Viktor Prasanna. 2014. GoFFish: A sub-graph centric framework for large-scale graph analytics. In *Euro-Par 2014 Parallel Processing*, Fernando Silva, Inłs Dutra, and Vtor Santos Costa (Eds.). Lecture Notes in Computer Science, Vol. 8632. Springer International Publishing, 451–462. DOI:http://dx.doi.org/10.1007/978-3-319-09873-9_38

George M. Slota, Kamesh Madduri, and Sivasankaran Rajamanickam. 2014. PULP: Scalable multi-objective multi-constraint partitioning for small-world networks. In *2014 IEEE International Conference on Big Data*. IEEE, Washington, DC, 481–490.

Isabelle Stanton. 2014. Streaming balanced graph partitioning algorithms for random graphs. In *Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'14)*. SIAM, 1287–1301. http://dl.acm.org/citation.cfm?id=2634074.2634169.

Isabelle Stanton and Gabriel Kliot. 2012. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'12)*. ACM, New York, NY, 1222–1230. DOI:http://dx.doi.org/10.1145/2339530.2339722

Philip Stutz, Abraham Bernstein, and William Cohen. 2010. Signal/Collect: Graph algorithms for the (semantic) web. In *Proceedings of the 9th International Semantic Web Conference on the Semantic Web - Volume I*. Springer-Verlag, Berlin, 764–780. http://dl.acm.org/citation.cfm?id=1940281.1940330.

Siddharth Suri and Sergei Vassilvitskii. 2011. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th International Conference on World Wide Web (WWW'11)*. ACM, New York, NY, 607–614. DOI:http://dx.doi.org/10.1145/1963405.1963491

Serafettin Tasci and Murat Demirbas. 2013. Giraphx: Parallel yet serializable large-scale graph processing. In *Proceedings of the 19th International Conference on Parallel Processing*. Springer-Verlag, Berlin, 458–469. DOI:http://dx.doi.org/10.1007/978-3-642-40047-6_47

Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From “think like a vertex” to “think like a graph.” *Proc. VLDB Endow.* 7 (2013), 3.

Charalampos Tsourakakis, Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Maria Tsiarli. 2013. Denser than the densest subgraph: Extracting optimal quasi-cliques with quality guarantees. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'13)*. ACM, New York, NY, 104–112. DOI:http://dx.doi.org/10.1145/2487575.2487645

Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. 2014. FENNEL: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining (WSDM'14)*. ACM, New York, NY, 333–342. DOI:http://dx.doi.org/10.1145/2556195.2556213

Johan Ugander and Lars Backstrom. 2013. Balanced label propagation for partitioning massive graphs. In *Proceedings of the 6th ACM International Conference on Web Search and Data Mining (WSDM'13)*. ACM, New York, NY, 507–516. DOI:http://dx.doi.org/10.1145/2433396.2433461

Leslie G. Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (Aug. 1990), 103–111. DOI:http://dx.doi.org/10.1145/79173.79181

Luis Vaquero, Félix Cuadrado, Dionysios Logothetis, and Claudio Martella. 2013. xdgp: A dynamic graph processing system with adaptive partitioning. *arXiv preprint arXiv:1309.1049*. http://arxiv.org/abs/1309.1049

Luis Vaquero, Felix Cuadrado, and Matei Ripeanu. 2014. Systems for near real-time analysis of large-scale dynamic graphs. *arXiv preprint arXiv:1410.1903*. http://arxiv.org/abs/1410.1903

Luis M. Vaquero, Felix Cuadrado, Dionysios Logothetis, and Claudio Martella. 2014. Adaptive partitioning for large-scale dynamic graphs. In *Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems (ICDCS'14)*. IEEE Computer Society, Washington, DC, 144–153. DOI:http://dx.doi.org/10.1109/ICDCS.2014.23

Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. 1992. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA'92)*. ACM, New York, NY, 256–266. DOI:http://dx.doi.org/10.1145/139669.140382

Guozhang Wang, Wenlei Xie, Alan J. Demers, and Johannes Gehrke. 2013. Asynchronous large-scale graph processing made easy. In *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR'13)*.

Lu Wang, Yanghua Xiao, Bin Shao, and Haixun Wang. 2014. How to partition a billion-node graph. In *2014 IEEE 30th International Conference on Data Engineering (ICDE'14)*. 568–579. DOI:http://dx.doi.org/10.1109/ICDE.2014.6816682

Peng Wang, Kaiyuan Zhang, Rong Chen, Haibo Chen, and Haibing Guan. 2014. Replication-based fault-tolerance for large-scale graph processing. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'14)*. 562–573. DOI:http://dx.doi.org/10.1109/DSN.2014.58

Rui Wang and K. Chiu. 2013. A stream partitioning approach to processing large scale distributed graph datasets. In *2013 IEEE International Conference on Big Data*. 537–542. DOI:http://dx.doi.org/10.1109/BigData.2013.6691619

Jim Webber. 2012. A programmatic introduction to Neo4J. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH'12)*. ACM, New York, NY, 217–218. DOI:http://dx.doi.org/10.1145/2384716.2384777

Jeremiah James Willcock, Torsten Hoefler, Nicholas Gerard Edmonds, and Andrew Lumsdaine. 2011. Active pebbles: Parallel programming for data-driven applications. In *Proceedings of the International Conference on Supercomputing (ICS'11)*. ACM, New York, NY, 235–244. DOI:http://dx.doi.org/10.1145/1995896.1995934

Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. 2015. Sync or async: Time to fuse for distributed graph-parallel computation. In *Proceedings of 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

Cong Xie, Ling Yan, Wu-Jun Li, and Zhihua Zhang. 2014. Distributed power-law graph computing: Theoretical and empirical analysis. In *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (Eds.). Curran Associates, 1673–1681. http://papers.nips.cc/paper/5396-distributed-power-law-graph-computing-theoretical-and-empirical-analysis.pdf.

Wenlei Xie, Guozhang Wang, David Bindel, Alan Demers, and Johannes Gehrke. 2013. Fast iterative graph computation with block updates. *Proc. VLDB Endow.* 6, 14 (Sept. 2013), 2014–2025. DOI:http://dx.doi.org/10.14778/2556549.2556581

Ning Xu, Lei Chen, and Bin Cui. 2014. LogGP: A log-based dynamic graph partitioning method. *Proc. VLDB Endow.* 7 (2014), 14.

Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014a. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proc. VLDB Endow.* 7 (2014), 14.

Da Yan, James Cheng, Kai Xing, Li Lu, Wilfred Ng, and Yingyi Bu. 2014b. Pregel algorithms for graph connectivity problems with performance guarantees. *Proc. VLDB Endow.*, 7 (2014).

Eiko Yoneki and Amitabha Roy. 2013. Scale-up graph processing: A storage-centric view. In *1st International Workshop on Graph Data Management Experiences and Systems (GRADES'13)*. ACM, New York, NY, Article 8, 6 pages. DOI:http://dx.doi.org/10.1145/2484425.2484433

Pingpeng Yuan, Wenya Zhang, Changfeng Xie, Hai Jin, Ling Liu, and Kisung Lee. 2014. Fast iterative graph computation: A path centric approach. In *Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*. IEEE Press, Piscataway, NJ, 401–412. DOI:http://dx.doi.org/10.1109/SC.2014.38

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, Berkeley, CA, 10–10. http://dl.acm.org/citation.cfm?id=1863103.1863113

ZengFeng Zeng, Bin Wu, and Haoyu Wang. 2012. A parallel graph partitioning algorithm to speed up the large-scale distributed graph mining. In *Proceedings of the 1st International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications (BigMine'12)*. ACM, New York, NY, 61–68. DOI:http://dx.doi.org/10.1145/2351316.2351325

Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15)*. ACM, New York, NY, 183–193. DOI:http://dx.doi.org/10.1145/2688500.2688507

Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. 2012a. Accelerate large-scale iterative computation through asynchronous accumulative updates. In *Proceedings of the 3rd Workshop on Scientific Cloud Computing Date (ScienceCloud'12)*. ACM, New York, NY, 13–22. DOI:http://dx.doi.org/10.1145/2287036.2287041

Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. 2012b. iMapReduce: A distributed computing framework for iterative computation. *J. Grid Comput.* 10, 1 (March 2012), 47–68. DOI:http://dx.doi.org/10.1007/s10723-012-9204-9

Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. 2013. PrIter: A distributed framework for prioritizing iterative computations. *IEEE Trans. Parallel Distrib. Syst.* 24, 9 (Sept. 2013), 1884–1893. DOI:http://dx.doi.org/10.1109/TPDS.2012.272

Yue Zhao, Kenji Yoshigoe, Mengjun Xie, Suijian Zhou, Remzi Seker, and Jiang Bian. 2014. LightGraph: Lighten communication in distributed graph-parallel processing. In *Proceedings of the 2014 IEEE International Congress on Big Data (BIGDATACONGRESS'14)*. IEEE Computer Society, Washington, DC, 717–724. DOI:http://dx.doi.org/10.1109/BigData.Congress.2014.106

Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. 2015. FlashGraph: Processing billion-node graphs on an array of commodity SSDs. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, Berkeley, CA, 45–58. http://dl.acm.org/citation.cfm?id=2750482.2750486

Xianke Zhou, Pengfei Chang, and Gang Chen. 2014. An efficient graph processing system. In *Web Technologies and Applications*, Lei Chen, Yan Jia, Timos Sellis, and Guanfeng Liu (Eds.). Lecture Notes in Computer Science, Vol. 8709. Springer International Publishing, 401–412. DOI:http://dx.doi.org/10.1007/978-3-319-11116-2_35

Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. 2008. Large-scale parallel collaborative filtering for the Netflix prize. In *Proceedings of the 4th International Conference on Algorithmic Aspects in Information and Management*. Springer-Verlag, Berlin, 337–348. DOI:http://dx.doi.org/10.1007/978-3-540-68880-8_32

Received January 2015; revised July 2015; accepted August 2015