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 standard 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.* 1, 1, Article 1 (January 2015), 35 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

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 two years [Kim et al. 2014], a figure that, because of the continual data deluge, was out of date by the time the report was published [IBM et al. 2011].

To address the challenges posed by the increased volume, velocity, and variety of 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 execu-

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.
Author's addresses: Robert Ryan McCune, 326 Cushing Hall, University of Notre Dame, Notre Dame, IN, 46656; email: rmccune@nd.edu; Tim Weninger, 353 Fitzpatrick Hall, University of Notre Dame, IN, 46656; email: tweninge@nd.edu; Greg Madey, 384 Fitzpatrick Hall, University of Notre Dame, Notre Dame, IN, 46556; email: gmadey@nd.edu.

DOI: http://dx.doi.org/10.1145/0000000.0000000

tion 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 can even result in poor performance in many cases [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 off of 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 multi-processor 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 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 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 after a fixed number of supersteps has occurred or when all vertices are inactive.

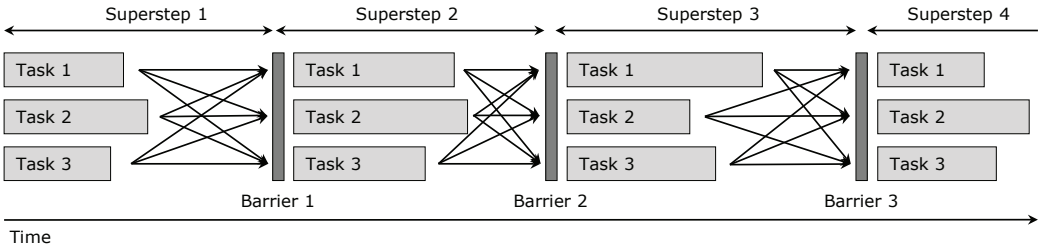


Fig. 1: Example of Bulk Synchronous Parallel execution with 3 tasks/workers over 4 supersteps. Each task may have varying durations after which messages are passed. The barriers control synchronization across the entire system.

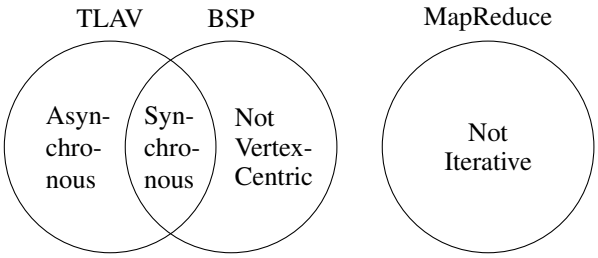


Fig. 2: Comparison of the Think Like a Vertex (TLAV), Bulk Synchronous Parallel (BSP), and MapReduce models of computation. TLAV frameworks employ a vertex-centric programming model, while BSP executes in synchronous rounds, or supersteps, with interspersed communication. The reduce phase of MapReduce/Hadoop is like one round of BSP/TLAV execution.

A comparison of TLAV frameworks, BSP and MapReduce 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. The reduce phase of MapReduce is like one round of BSP/TLAV synchronous execution. A more in-depth comparison between MapReduce and BSP may be found in [Pace 2012] and references therein. Both Pregel and MapReduce are designed for distributed architectures on commodity machines, abstracting away lower-level details of distributed system development. Pregel, MapReduce, and BSP all adopt the conventional master/worker architectural pattern.

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 the concepts and components, and makes these other contributions:

(1) Analyzes 4 principle components of TLAV frameworks, identifying the trade-offs and providing data-driven discussion
(2) Categorizes foundational TLAV frameworks, including single-machine, temporal, and alternative frameworks
(3) Exposes trade-offs between locality and computational perspectives

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 vertex-centric frameworks. Section 4 categorizes TLAV implementations. Section 5 discusses related work. Finally, Section 6 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, inter-machine communication, termination, check-pointing, etc. 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, 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 data sets [Low et al. 2010].

Many graph problems can be solved by both a sequential, shared-memory algorithm as well as 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 local data; as a result they are highly scalable, and inherently parallel with relatively low inter-machine communication. For example, runtime

Algorithm 1: Single Source Shortest Path for a Synchronized TLAV Framework

```
input: A graph (V, E) = G with vertices v ∈ V and edges from i → j s.t. e_ij ∈ E,
       and starting point vertex v_s ∈ V
foreach v ∈ V do shrtest_path_len_v ← ∞;    /* initialize each vertex data to ∞ */
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 ∈ 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← min(receive (path_length));
        /* set current vertex-data to minimum value                    */
2       if minIncomingData < shrtest_path_len_v then
3           shrtest_path_len_v ← minIncomingData;
4           foreach e_vj ∈ E do
                /* send shortest path + edge weight to outgoing edges    */
5               path_length ← shrtest_path_len_v+weight_e;
6               send (path_length, j);
7           end
8       else
            /* Nothing is sent, vertex deactivated                       */
9           halt ();
10      end
    end
until no more messages are sent;
```

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, re-usable 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 Dijkstra's 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 are have a superlinear time complexity: Djisktra's runs in $O(|E| \log |E| + |V|)$ and Bellman-Ford's runs in $|E| \times |V|$, where $|E|$ 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 $O(|V|^2)$ for both variants.

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 Alg. 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 to write the inner-portion of Alg. 1 denoted by line numbers; the outermost loop and the parallel execution is handled by the framework. Because lines 1-10 are executed on the each vertex these lines are known as the *vertex program*.

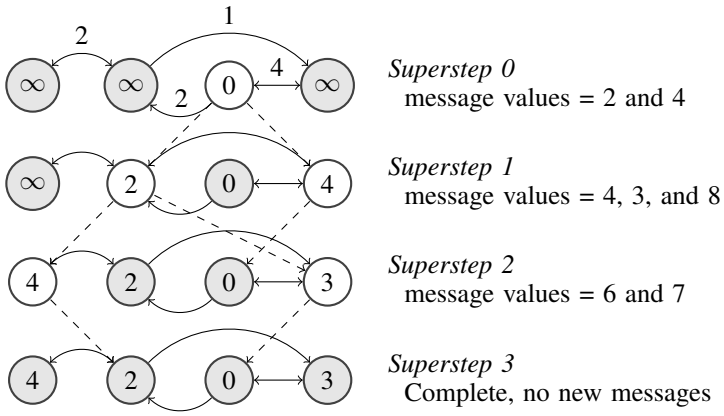


Fig. 3: Computing the Single Source Shortest Path in a graph. Dashed lines between supersteps represent messages (with values listed to the right), and shaded vertices are inactive. Edge weights pictorially included in first layer for Superstep 0, then subsequently omitted.

Figure 3 depicts the execution of Alg. 1 for a graph with 4 vertices and 6 weighted directed edges. Only the source vertex begins in an active state. In each superstep, a vertex processes its incoming messages, 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.

3. FOUR PILLARS OF TLAV FRAMEWORKS

A TLAV framework is software that supports the iterative execution of a user-defined vertex programs 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 principle 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) Model of Computation - Implementation of vertex program execution
(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

[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 4.2, while some TLAV frameworks are implemented for a single machine without the specific intention of developing for a non-distributed 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 and Section 4.1] or GRACE [see Section 3.1.3]), the single-machine frameworks presented in Section 4.2 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.

The discussion proceeds as follows: the timing policy of vertex programs is presented in Subsection 3.1, where system execution can be synchronous, asynchronous, or hybrid. Communication between vertex programs is presented in Subsection 3.2, where intermediate data is shared primarily through message-passing or shared-memory. The implementation of vertex program execution is presented in Subsection 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 Subsection 3.4.

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 inter-relation 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 a paper. 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 is 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 bulk synchronous parallel (BSP) processing model discussed above. 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 trade-offs.

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 straight-forward 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. 2014], so system throughput must remain high to justify the cost of synchronization, since such

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

Table I: Execution timing model of selected frameworks.

coordination can be relatively costly. However, when the number of active vertices drops or the workload amongst workers becomes imbalanced, system resources can become under-utilized. 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 than 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 others' 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], albeit 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*. demonstrated that, for PageRank, the majority of vertices converged within one superstep, while only 3% of vertices required more than 10 supersteps [Low et al. 2012]. 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 sub-optimal 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; Han 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, which is 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 challenges development as framework users may have to consider low-level concurrency issues [Wang et al. 2013], like in GraphLab when a user 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. 2014]. 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 reduces the number of supersteps by decoupling intra-processor computation from the inter-processor communication and synchronization [Chen et al. 2014]. 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 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 MPI-style messages in the standard model. A similar model is adopted by the P++ framework [Zhou et al. 2014].

GRACE is a framework that combines the scalability advantages of synchronous design with asynchronous execution [Wang et al. 2013]. In a synchronous model, each vertex gets executed once per round in fixed order, but in asynchronous systems, any active vertex is eligible for execution and can be dynamically scheduled. The single-machine framework GRACE explores dynamic scheduling of vertices from within a single synchronous round. 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 amongst 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 the 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 4a, and Figures 4b-4e depict 4 TLAV communication implementations of the sample graph. For each implementation, vertices are partitioned across 2 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 4d and 4e, where the graph is cut along vertex C). Solid arrows represent local[2] communication and dashed arrows represent network traffic.

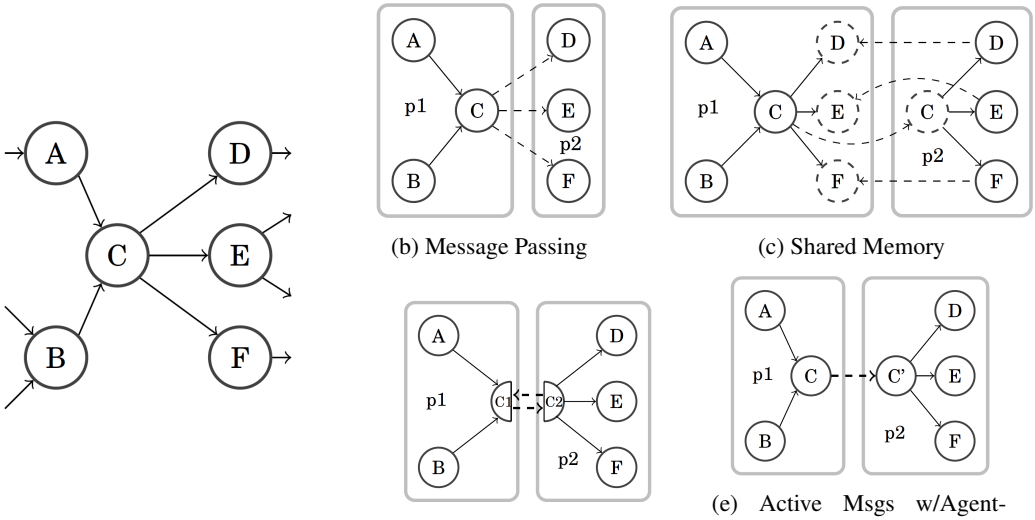
3.2.1. Message Passing. In the message passing method of communication, also known as the LOCAL model of distributed computation [Peleg 2000], information is sent from one vertex program kernel to another via a message. A message contains local vertex data and is addressed to the ID of the recipient vertex. In the archetypal message-passing framework Pregel [Malewicz et al. 2010], a message can be addressed anywhere, but because vertices do not have ID information of all of 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. Else, 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

[2]internal to a single machine

[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) 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).

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 4b, where vertex C sends (an) inter-machine 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 routing is omitted from the figure without loss of generality. Figure 4b 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 5c.

3.2.2. Shared Memory. Shared memory exposes vertex data as shared variables that can be directly read or 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 4.2), 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. Inter-machine communication for distributed shared memory still occurs through network messages. The Trinity framework [Shao et al. 2013a] 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 local 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, like 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].

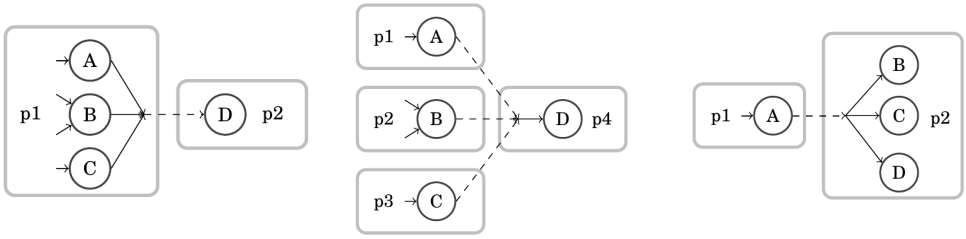
Shared memory implementations are depicted in Figure 4c and Figure 4d. In Figure 4c, ghost vertices, represented by dashed circles, are created for every neighboring vertex residing on a remote machine, as implemented by GraphLab [Low et al. 2012]. One disadvantage of shared-memory frameworks is seen 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 4d 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 4d, 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: (i) both $C1$ and $C2$ compute a partial result based on local[4] neighbors, (ii) the partial result is sent over the network from the mirror $C2$ to the master $C1$, (iii) the master computes the final result for the iteration, (iv) the master transmits the result back to the mirror over the network, then (v) the result is sent to local neighbors as necessary. PowerGraph demonstrates how the combination of advanced components, *i.e.*, 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. Significant deterioration in performance was noted in [Han et al. 2014; 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]. Active messaging is a way of bringing computation to data, where a message contains both data as well as the operator to be applied to the data [von Eicken et al. 1992]. Active messages 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 intermediate state, like message queues or edge data. When combined with the framework's novel Agent-Graph model, described below, GRE demonstrates 20%–55% reduction in runtime compared to PowerGraph across three benchmark algorithms in real and synthetic datasets, including 39% reduction in the execution time per iteration for PageRank on the Twitter graph when scaled across 192 cores over 16 machines

[4]same partition (see Section 3.4), or basically intra-machine



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

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

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 inter-machine messaging. Figure 4e 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 5c. 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 5a. 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 the network between machines. 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 5b).

The combiner or *sender-side aggregation*, 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, then send the new sum to v. A Combiner can especially reduce network traffic when v is remote (Figure 5a). When v is local, a combiner can still reduce memory overhead by aggregating messages before placement into the incoming message queue (Figure 5b). For example, for the single-source shortest path algorithm, a combiner implementation resulted in a four-fold 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 5c. The strategy has been employed in multiple frameworks, including the *Large Adjacency List Partitioning* in GPS [Salihoglu and Widom 2013], IBM's X-Pregel [Bao and Suzumura 2013], as the *fetch-once* behavior in LFGraph [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 out-going 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 existance of an optimal vertex-degree threshold. In X-Pregel, a ten-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 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, albeit 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.

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, like in the 1/2-approximation for maximum weight matching [Preis 1999], the Edge Cleaning on Demand (ECOD) optimization only deletes stale edges when, counter-intuitively, activity is requested for the stale edge [Salihoglu and Widom 2014]. To avoid slow convergence, ECOD is only employed above a certain threshold, *e.g.*, when more than 1% of all vertices are active. Both SEAS and ECOD exploit a trade-off between sending messages proportional to the number of vertices or proportional to the number of edges. Another strategy for reducing communication, based on *aggregate computation*, is discussed in Section 3.3.3.

3.3. Computation

Having discussed timing and communication tradeoffs in TLAV frameworks, we turn out attention to models of vertex computation. First, different implementations of vertex programs are introduced, then the push and pull operators of computation, i.e., how information flows in the system relative to computational operators, is discussed.

3.3.1. Vertex Programming Models. Different TLAV frameworks have implemented the vertex program abstraction with either one, two, or three phases of execution, along with an *edge-centric* variant. Different implementations of the vertex programming model introduce opportunities to interact with other system components that collectively influence performance.

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, *e.g.*, 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, programed 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], and is present in GraphChi as the Standard Programming Model [Kyrola et al. 2012]. 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 occurs most commonly in TLAV systems where data is read/written to/from edges.

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 as well as the operator to be executed on the data [von Eicken et al. 1992]. In the first phase of the model, messages are both 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 (i) Gather-Apply, and (ii) Scatter, the Scatter-Combine model uses active messages to institute (i) Scatter-Gather, and then (ii) Apply. The GRE framework combines Scatter-Combine with a novel representation of the underlying data graph, called the Agent-Graph, described above, 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 is the second generation of the shared-memory GraphLab package, and supports both synchronous and asynchronous communication.

PowerGraph incorporates the GAS model with vertex-cut partitioning, where partitioning is edge-centric and vertices are mirrored across machines (see Section 3.4.3). With a three-phase programming model, each copy of a duplicated vertex can independently perform a local gather phase, send the result to a designated master-copy of the vertex to perform the apply phase, and finally the master sends the result back with the mirrors which execute a local scatter. The GAS model with vertex-cuts better accommodates scale-free graphs, where communication and workload can become substantially imbalanced for high-degree vertices.

Edge-Centric. The X-Stream framework provides an *edge-centric* Scatter-Gather programming model [Roy et al. 2013], as opposed to a vertex-centric programming model. In the vertex-centric Scatter-Gather model, the Scatter and Gather functions take a vertex as input, and accesses the incident edges of the vertex by way of sequential access to a sorted list of edges. Although, in X-Stream, the Scatter and Gather functions take an edge as input, the overall framework is still vertex-oriented because the computations within the edge-centric Scatter and Gather functions still operate on the source or target vertex data. X-Stream is a synchronous, shared-memory framework designed for a single machine, and explores the increased bandwidth trade-off offered by streaming edges rather than randomly reading edges from disk. Higher bandwidth is therefore achieved in X-Stream because edges are read sequentially from storage rather than via random lookups that are often the result of vertex-centric approaches. More details on the X-Stream system are discussed in Section 4.2.

3.3.2. Push vs. Pull. The flow of information for vertex-program execution can be generally characterized by two different modes, a *push* mode and *pull* mode [Nguyen et al. 2013; Han et al. 2014; Cheng et al. 2012; Han et al. 2014]. In *push* mode, information flows from the active vertex executing the program out to the vertex's neighbors, like when the vertex computes a new value and sends

messages along outgoing edges. In *pull* mode, information flows from neighboring vertices into the active vertex, as in GraphLab when a vertex reads values of in-edge neighbors from shared memory in order to execute an update.

Push and pull modes are commonly associated with databases and transactional processing, so are more often addressed by TLAV frameworks as part of a more complete graph management system, which may include graph databases (see Related Work in Section 5) or temporal TLAV frameworks (see Section 4.3) [Cheng et al. 2012]. Within TLAV frameworks, the implementation of a push or pull mode may be an explicit design choice [Nguyen et al. 2013], or naturally arise from other design decisions. Message passing echoes information flow of the push model, but messaging would appear inefficient for the pull model. Both modes support asynchronous execution, while the push mode allows for sender-side aggregation [Cheng et al. 2012].

Push or pull modes may also arise out the operation performed by the vertex program. A recently developed shared-memory breadth-first search algorithm achieves remarkable performance by switching between push and pull modes of exploration [Beamer et al. 2013]. Inspired in part by this approach is Ligra, a single-machine shared-memory framework that supports vertex-oriented programs for graph traversals within a global sequential context [Shun and Blelloch 2013]. Ligra allows programs to switch between push and pull-based operators based on a specified threshold. Galois is a graph processing infrastructure that similarly includes the vertex programming model [Kulkarni et al. 2009], which also supports both push and pull modes, depending on algorithmic implementation [Nguyen et al. 2013].

The choice of mode has special implications to temporal graph processing, where changes in vertex values can arise from mechanisms other than update computations. The Kineograph and Chronos temporal frameworks support both push and pull modes [Han et al. 2014; Cheng et al. 2012], and Chronos further tests how push and pull modes impact caching [Han et al. 2014].

3.3.3. Optimizations. Two optimization methods improve runtime by operating on the graph as a whole in addition to the vertex-oriented computing model. 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.

Similarly, the Single Pivot (SP) optimization [Salihoglu and Widom 2014], first presented in [Quick et al. 2012], also performs an aggregate computation. For algorithms that execute breadth-first search (BFS) across all vertices, *e.g.*, 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, 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.

The integration of graph-level and vertex-level processing echoes the combination of top-down and bottom-up approaches utilized for fast, shared memory BFS in [Beamer et al. 2013]. More radical optimizations can be characterized as alternative computational models, such as partition- or neighborhood-centric programming models, and are presented in Section 4.4.

3.4. Partitioning

For TLAV frameworks, a well-partitioned graph often results in improved performance [Salihoglu and Widom 2013]. Effective partitioning of a graph aims to evenly distribute vertices across machines for a balanced workload, while minimizing the number of edge cuts between machines in order to avoid costly network traffic. This objective is formally known as the *k-way graph partitioning* problem, and is NP-complete with no fixed-factor approximation [Andreev and Racke 2006; Meyerhenke et al. 2014]. Leading work 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 multi-level partitioning approach [Abou-Rjeili and Karypis 2006]. Deviations are often allowed between sizes of partitions in the form of a "slackness" parameter in exchange for better cuts [Karypis and Kumar 1996]. Although it is generally accepted and experimentally validated that TLAV frameworks benefit from a well-partitioned graph, one experimental result demonstrates otherwise [Shao et al. 2013b] by claiming that high local messaging stemming from quality partitioning can become a processing bottleneck. For environments, such as the cloud where bandwidth between machines may be uneven, the Surfer framework [Chen et al. 2012] presents a bandwidth-aware partitioning approach.

Graph partitioning with the METIS family of graph and hypergraph 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, METIS-partitioning and other heuristics are impractical for graphs at medium and large scales, because of high computational costs and the global random access required of all vertices. Even parallelized METIS, ParMETIS, requires this global view which challenges processing when graphs don't fit in memory. For graphs of even medium-size, alternative partitioning methods have been developed that achieve well-balanced workloads and reduced communication without needing global, random access to the entire graph. Current partitioning approaches for TLAV frameworks are characterized as distributed heuristics 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 that require little or no centralized coordination, yielding a partitioning that's bottom-up rather than top-down. Distributed partitioning is related to distributed community detection in networks [Gehweiler and Meyerhenke 2010; Ramaswamy et al. 2005] with two main differences being: 1) communities can overlap whereas partitions cannot, and 2) community detection does not typically require *a priori* specification of the number of communities whereas partitioning methods clearly do. Much distributed partitioning work has been inspired by distributed community detection methods, 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 non-local 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 the the cost of random vertex access on a distributed architecture, as needed 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 multi-level partitioning scheme, which processes the partitioning in blocks to accommodate multi-level 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 multi-level 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 as the graph is loaded onto the cluster from storage, as opposed to performing distributed

partitioning after the graph is already loaded. The partitioning is performed by a program called the graph loader that runs on the master during initialization. The accepted streaming model assumes a single, centralized graph loader that reads data serially from disk and decides where to place the graph data amongst the 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 concurrency between the parallel partitioners would likely be required, depending on the heuristic [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 graph loader that supports an extensive variety of graph loading-related processing tasks, in addition to partitioning.

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 algorithms in the streaming model 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, and is 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 data sets. 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 linear deterministic greed (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, and has been implemented by the PowerLyra framework [Chen et al. 2013] and an updated version of GraphLab [Tsourakakis et al. 2014]. Both FENNEL and LDG are 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 are also capable of partitioning in the presence of multiple constraints and in parallel without inter-stream communication.

3.4.3. Vertex Cuts. A vertex-cut, depicted in Figure 4d, 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 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 was the first TLAV framework to implement vertex-cuts; it combines vertex-cuts with the three-phase GAS computational 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 vertex-cuts demonstrated an 8-fold decrease in the platform's communication cost. GraphBuilder [Jain et al. 2013], the open-source graph loader, supports vertex-cuts and implemented grid and torus-based vertex-cut strategies that were later included in PowerGraph. PowerLyra [Chen et al. 2013] 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]. 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 outperforming 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-centered 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 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. The number of active vertices can change drastically over the course of execution, which will imbalance processing and increase run time. Dynamic repartitioning was developed to balance computation by migrating vertices between workers during processing. Much of the current work on dynamic repartitioning in TLAV frameworks is based on earlier work that was previously implemented for the processing of dynamic meshes [Schloegel et al. 2001]. TLAV frameworks may opt to repartition the graph from scratch using a one-pass streaming algorithm or a restreaming algorithm [Nishimura and Ugander 2013], but such an approach would take too long or ignore relevant information. Dynamic repartitioning has been implemented for distributed graph databases to reduce query latency [Yang et al. 2012; Labouseur et al. 2014], but

Framework	Dynamism	Reassignment	Relocation	Coord.	Balance
GPS	Algorithm	Sent Msgs	Brdcst Vert ID	Decent.	Swap Min-Set
Mizan	Algorithm	Sent/Recv Msgs Run Time	Distr. Hash Table	Decent.	Metric-based Swap
XPregel	Algorithm	Sent/Recv Msgs	Brdcst New Worker	Cent.	Repartition Largest Worker
xDGP	Topology	Neighbor Labels	Brdcst New Worker	Decent.	Capacity Fraction
LogGP	Both	Runtime	Lookup Table	Cent.	Repartition Longest Workers
Catch the Wind	Algorithm	Sent/Recv Msgs	Lookup Table	Decent.	Quota

Table II: Feature summary for TLAV frameworks supporting dynamic repartitioning.

vertices can be replicated and most queries only process a small number of nodes instead of the entire graph. A survey of large-scale dynamic graph systems discussing the difference between partitioning and dynamic replication including TLAV frameworks is presented in [Vaquero et al. 2014].

Previous partitioning strategies presented in this section apply to the initial partition of the graph across the cluster, but dynamic partitioning reassigns vertices to different workers during the execution of an algorithm in between vertex updates. Dynamic repartitioning is beneficial when the number of active vertices changes during processing, and the cost of balancing computation by transferring vertices over the network will be offset by an overall reduction in runtime. The number of active vertices may change by nature of the algorithm, like in a multi-phase algorithm such as maximal independent set [Shang and Yu 2013]; active vertices may also change due to graph mutations, like the addition or deletion of vertices. Changes in the number of active vertices for different classifications of algorithms is explored in [Shang and Yu 2013].

According to [Salihoglu and Widom 2013], a dynamic repartitioning strategy must address three aspects, i) how to select vertices to reassign, ii) how and when to move the assigned vertices, and iii) how to locate the reassigned vertices. Other properties of a strategy include whether worker coordination is centralized or decentralized, and how vertex balance across workers is enforced. Densification, akin to the rich-get-richer phenomenon, can occur in greedy or decentralized protocols for partitioning or clustering, where one partition becomes over-populated as the repeated destination for migrated vertices [Vaquero et al. 2013]. In response, such protocols often implement constraints that prevent a partition from exceeding a certain capacity; for example, the XPregel framework avoids densification by only allowing the worker with most vertices and edges to migrate vertices [Bao and Suzumura 2013]. Table II presents the features for the 6 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].

Among the 6 frameworks that implement dynamic repartitioning, all are synchronous, and the dynamic repartitioning process occurs at the end of a superstep separate from vertex computation. 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].

Computing which vertices to migrate and then transmitting the information over the network incurs a significant cost, and improves overall runtime only under select circumstances [Salihoglu and Widom 2013]. Unlike other forms of partitioning that can significantly improve performance, many experiments with dynamic repartitioning demonstrate little or no improvement. Results in [Bao and Suzumura 2013] show that while network I/O is significantly reduced over time, overall runtime

has only minor improvements. Dynamic repartitioning for GPS is detrimental for all tests 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 effective for processing workloads where active vertices fluctuate, but in the presented PageRank implementation, all vertices are always active, so dynamic repartitioning performs predictably poor. Still, even under appropriate dynamic circumstances, performance does not significantly improve. Nevertheless, dynamic repartitioning should only be considered for processes with a highly dynamic number active vertices.

4. FRAMEWORKS

This section categorizes the various implementations of TLAV components presented in the previous section. The general but foundational TLAV frameworks are presented in Section 4.1, frameworks that employ novel methods to process large graphs on a single machine are presented in Section 4.2, TLAV processing of dynamic graphs is presented in Section 4.3, and sub-graph-centric computational models that are related to TLAV processing is presented in Section 4.4.

4.1. Fundamental TLAV Frameworks

Pregel, GraphLab, and GraphX for Spark are three families of frameworks consistently identified in the literature as theoretically formative and widely used in practice. Pregel employs a synchronous message-passing model, GraphLab an asynchronous shared-memory model, PowerGraph, being the second-generation of GraphLab, incorporates innovative components to address shortcomings in power-law graph processing, and GraphX is a vertex-centric library developed on-top of the widely used Spark framework for Big Data processing.

Pregel. The first TLAV framework introduced in the literature is Pregel, named after the Pregel River from the Euler's Seven Bridges of Konigsberg problem. Developed at Google, Pregel was first described in a research blog post [Czajkowsi 2009], then further detailed in a seminal 2010 paper [Malewicz et al. 2010]. Pregel is a synchronous, message-passing framework based on the BSP model for parallel computing [Valiant 1990]. An example Pregel algorithm and execution is provided in Section 2.1. Since the introduction of Pregel, numerous frameworks have been developed that not only depart from the synchronous, message-passing model (*e.g.*, GraphLab), but also introduce optimizations into the Pregel model.

Several other frameworks have implemented Pregel with further enhancements or optimizations. Most notably, GPS [Salihoglu and Widom 2013], developed in Java, includes dynamic repartitioning, a receiver-side scatter scheme called Large Adjacency List Partitioning, and the ability to perform global computations. A Scala-based Pregel implementation includes graph construction and avoids inversion control [Haller and Miller 2011], and another C++ implementation, Pregel+, provides vertex mirroring and a request-respond interface [Yan et al. 2014b]. Kylin [Ho et al. 2013] includes pull messaging, lazy vertex loading, and vertex-weighted partitioning; Surfer performs bandwidth-aware partitioning [Chen et al. 2012]; Rondero [Macambira and Guedes 2010] supports the temporary spilling of message queues on to out-of-core storage; and Seraph [Xue et al. 2014] allows multiple jobs to be run in parallel on the same graph with some restrictions [Vaquero et al. 2014].

GraphLab. A compliment to Pregel is the GraphLab framework, an asynchronous, shared-memory TLAV framework developed in C++ and licensed as open source. GraphLab was first developed as a single-machine framework [Low et al. 2010], which evolved into a distributed version 2 years later [Low et al. 2012]. PowerGraph is the next iteration of GraphLab [Gonzalez et al. 2012], which implements vertex cuts from Section 3.4.3 and the GAS model of computation in Section 3.3.1 to meet the computational demands posed by high degree vertices of power-law graphs. The developers of GraphLab and PowerGraph have also developed a single-machine version framework called GraphChi [Kyrola et al. 2012]. These products, collectively, form the foundation of the

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	Semi-External Memory with Page Cache	[Zheng et al. 2013]
PathGraph	Disk/SSD	Compressed DFS Traversal Trees	[Yuan et al. 2014]

Table III: Single Machine Frameworks

GraphLab Company and its product GraphLab Create, which has python bindings, but is closed source[5].

GraphX on Spark. Spark is a Scala-based, open source distributed processing framework for Big Data that utilizes Resilient Distributed Datasets (RDDs), an immutable, fault-tolerant, distributed memory abstraction, to perform general distributed computation [Zaharia et al. 2010; Zaharia et al. 2012]. The GraphX library has been developed on top of Spark through a series of data-parallel operators that collectively offer a Pregel-like API [Gonzalez et al. 2014].

4.2. Single Machine Architectures

While distributed frameworks abstract away the lower-level details of distributed processing, such environments also stipulate the availability of elaborate infrastructure, cluster management, fault tolerance, and performance tuning, in addition to the programmatic challenges of developing in a distributed environment. In contrast, single machines are easier to program and manage, but can't retain the entire graph in memory. This section overviews single machine frameworks that employ *novel* methods to, in a vertex-centric manner, process a graph beyond the size of the machine's memory. The main features of the 4 single machine frameworks 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 large-scale graph processing can still be performed on a single machine, considering moderate to high-end servers offer 100GB to 1TB of memory or more, which is enough capacity for many real and synthetic graphs reported in the literature. Such claims are supported by application; for instance the recommendation service at Twitter [Gupta et al. 2013], which implements a single machine graph processing system with 144 GB of RAM, finds that in practice one edge occupies roughly five bytes of RAM on average. Yet, such graphs are not practical on lower-end machines containing around 8 to 16 GB 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. At the core of 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 out-going edge adjacency sets, allowing for the fast look-up 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

[5]at the time of this writing the GraphLab Company is planning an open source version of their GraphLab Create product

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] and serves as the foundation for GraphLab Create.

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 trade-off 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 a three-phase Scatter-Gather programming model that reads input from, and writes updates 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. 2013], 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 semi-external 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 XStream, FlashGraph outperformed both by orders of magnitude even when the data for GraphChi and XStream was placed into RAM-disk [Zheng et al. 2013].

PathGraph. In addition to the path-centric programming model, further discussed in Section 4.4, 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.

4.3. Vertex-Centric Processing of Dynamic Graphs

Vertex-oriented frameworks typically process a static graph, but with the increasing wealth of dynamic activity as in telecommunication and social networks much insight can be gained by analyzing dynamic networks [Kostakos 2009; Holme and Saramäki 2012]. Several sequential graph algorithms have been developed to capture these rich temporal properties [Ning et al. 2007; Leskovec et al. 2007], but few TLAV frameworks are capable of analyzing dynamic networks because TLAV processing is typically performed offline and in batch. In response, some frameworks have been developed to date for performing vertex-centric temporal graph processing.

This section overviews 4 frameworks, presented in Table IV, and discusses how the frameworks relate to the key ideas in dynamic graph processing. Generalized temporal graph analysis is a rich, complex, and active subject of study, and a thorough review of the concepts underlying dynamic networks in general is beyond the scope of this paper. A thorough survey of dynamic graph algorithms

Framework	Processing Paradigm	
Kineograph	Snapshots	[Cheng et al. 2012]
Chronos	Temporal Locality	[Han et al. 2014]
GraphInc	Incramental	[Cai et al. 2012]
GoFFish	Subgraph-Centric Snapshots	[Simmhan et al. 2014b]

Table IV: Temporal Frameworks

is presented in [Aggarwal and Subbian 2014], dynamic processing framework requirements are discussed in [Khurana 2013; Fard et al. 2012], and an refined survey of more general dynamic graph analysis systems, which includes one TLAV framework discussed below, is presented in [Vaquero et al. 2014].

A dynamic graph encompasses evolutionary, temporal, or streaming changes of a graph over time, depending on the application instance. This means that the nodes and edges of the graph can be added or deleted, and their data properties modified, over time. The graph may be continually changing in real-time, or all changes may have already occurred, with timestamps associated with any given modification. Some models of dynamic graphs make assumptions regarding the nature of the network dynamism, like stipulating that vertices are only added, or that property data changes more frequently than the topology [Simmhan et al. 2014b].

Temporal analysis, such as how the diameter of the graph changes over time [Leskovec et al. 2007], attempts to capture dynamic properties of the dynamic graph, as opposed to static properties like just the graph diameter (at a graph instance). Algorithms for dynamic graphs typically utilize *snapshots*, or an instance of the graph at a particular moment in time, and then produce a quantity based on the processing of several graph snapshots.

Dynamic network analysis can be performed online or offline. Online processing characterizes low-latency network queries in response to network changes. An example online query is finding the immediate neighbors of a node, or determining if two nodes are connected. Response is fast because few vertices need be considered and is supported by a processing framework other than vertex-oriented computing. Vertex-centric processing is not typically associated with online queries, although some graph engines provide support for both vertex-centric processing and online queries [Shao et al. 2013a]. An online vertex-centric program would likely be a local distributed algorithm, where the runtime is constant and independent of network size (see [Suomela 2013] for a recent survey).

Kineograph. One such online, vertex-centric platform is a distributed system for processing streaming graph updates called Kineograph [Cheng et al. 2012]. This system is divided into roughly three components: 1) A front-end that ingests streaming graph data and provides update transactions to the graph. 2) Graph nodes, which perform the processing, which is separate from the graph storage. 3) A *snapshooter* that facilitates graph snapshots.

Chronos. Dynamic, iterative graph analysis is taken a step further in the Chronos system, which introduces certain processing optimizations [Han et al. 2014]. The principle that underlies the Chronos optimizations is the design choice of temporal locality instead of structural locality. Structural locality relates to the storage and processing of the graph in regards to structure where vertices are stored next to neighboring vertices within the same snapshot, and snapshots are processed independently of one another as in Kineograph. Chronos favors temporal locality, where different data for the same vertex present in multiple snapshots is stored together, and snapshots are iteratively processed around temporally varied vertices and edges using locality-aware batch scheduling. The temporal locality optimizations utilized by Chronos result in a 3 times speedup over snapshot-by-snapshot computation [Han et al. 2014].

GraphInc. Another technique for processing large dynamic datasets is called *incremental computation*, where only relevant portions of data are updated in response to small modifications. GraphInc is a framework that utilizes memoization to save state and incrementally process a graph [Cai et al.

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. 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	N	N	[Shun and Blelloch 2013]

Table V: Alternative Frameworks

2012]. GraphInc introduces a general framework for converting any vertex-oriented algorithm to an incremental algorithm, and demonstrates the effectiveness of incremental computation for micro-updates [Cai et al. 2012].

GoFFish. The snapshot technique exemplified in Kineograph is also adopted by GoFFish, a subgraph-centric framework that will be further discussed in Section 4.4. In [Simmhan et al. 2014b], GoFFish is augmented with storage optimizations and design patterns modeled after temporal execution patterns in order to process temporal graphs. Like Kineograph, snapshots are processed sequentially utilizing spatial locality, but unlike Kineograph, a program kernel from one snapshot can message a program kernel in the first superstep of a subsequent snapshot. The framework also introduces several general temporal execution patterns that may be utilized for additional storage optimizations.

4.4. Alternative Computation

Since the introduction of frameworks that utilize the vertex-centric programming model for graph processing, several related frameworks have been developed that adopt a programming model of coarser granularity, including sub-graph-, path-, and vertex subset-centric processing, summarized in Table V. This section further explores the adoption of alternative perspectives for graph processing.

Subgraph-Centric computation. A commonly implemented alternative programming model is based an iterative computation of subgraphs. Subgraph-centric computation has been implemented in two distinct ways. One method retains the vertex-centric programming model, but executes supersteps in two phases, where in the first phase, messages are exchanged over the network between partitions, and in the second phase, vertices within a partition execute the vertex function to completion, exchanging messages in memory. This method is observed in the GraphHP [Chen et al. 2014] and P++ [Zhou et al. 2014] frameworks, which decreases network communication and the number of supersteps, reducing synchronization overhead.

The second subgraph-centric approach allows for sequential algorithms to be executed on a subgraph, with messages exchanged between subgraphs, possibly over the network if on different machines. In the GoFFish framework [Simmhan et al. 2014a], subgraphs are connected, multiple subgraphs may reside on a partition, and messaging may occur from a subgraph to other subgraphs or specific non-local vertices.

Giraph++ [Tian et al. 2013] and Blogel [Yan et al. 2014a] implement both subgraph-centric approaches, where a subgraph function may be either vertex-centric or sequential. In Giraph++, messages between subgraphs occur between vertices on the subgraph boundaries, whereas, Blogel, like GoFFish, supports messaging from a subgraph to another subgraph or a specific non-local vertex.

Two other frameworks adopt different subgraph-centric approaches than the two previously presented. A version of the GRACE framework [Wang et al. 2013], which originally was designed to dynamically re-order synchronized vertex computation within a super-step, was developed into block-based GRACE [Xie et al. 2013]. Block-based GRACE divides a graph into subgraphs and iteratively executes block updates, while retaining the vertex-centric programming model. Execut-

ing vertex updates on a block basis improves locality and cache hits while reducing memory access time, which is a bottleneck for computationally light algorithms like PageRank.

PathGraph. A more specific type of subgraph, a traversal tree, is used in PathGraph as a programming model [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 traverse 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 (*c.f.*, Section 4.2) [Yuan et al. 2014].

Ligra. A vertex subset interface is implemented in Ligra [Shun and Blelloch 2013]. This framework is for single machines with an in-memory graph, arguing that high-end servers are appropriate for processing graphs of scale. 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. The framework does not implement storage techniques to enable large-scale processing on a single machine, so is included in this section and not Section 4.2. Vertex-centric algorithms can be implemented in Ligra. Ligra demonstrates how alternating between two processing perspectives can improve performance.

5. RELATED WORK

TLAV frameworks are tangentially related to graph processing, Big Data programming models, graph databases, and distributed algorithms, among others. Several graph processing frameworks have been recently developed sans the vertex-centric programming model. PEGASUS combines the BSP iterative computation 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 [Bulučc 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]. Several GPU-based TLAV implementations are presented in [Gharaibeh et al. 2013; Zhong and He 2014; Che 2014; Nurvitadhi et al. 2014; Fu et al. 2014].

MapReduce is a popular Big Data programming model[Dean and Ghemawat 2008; Polato et al. 2014], but does not immediately address i) iterative processing, or ii) 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 data and thus has no loop-awareness. Moreover, I/O is read/written to/from a file system, e.g. HDFS, which accessing for the potential intermediate state of iterative execution would be inefficient [Polato et al. 2014]. Still, several frameworks have extended MapReduce to support iterative execution using various work-arounds [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 generally acknowledged to be lacking [Cohen 2009; Malewicz et al. 2010]. Despite these shortcomings, some argue MapReduce should remain the sole "hammer" for Big Data analytics because of the widespread adoption throughout industry [Lin 2013]. A comparison of MapReduce and BSP is provided in [Kajdanowicz et al. 2014].

Both TLAV frameworks and graph databases, such as Neo4j [Webber 2012], HyperGraphDB [Iordanov 2010], and GBASE [Kang et al. 2011], treat vertices as first class citizens, and also face similar problems like graph partitioning. The key distinction is that databases focus on transactional processing while TLAV frameworks focus on batch processing [Chen et al. 2012]. Databases offer local queries, like finding a node's 1-hop neighbors, whereas TLAV frameworks process the whole

graph. Some more general graph management systems, such as Trinity [Shao et al. 2013a] and Grace [Prabhakaran et al. 2012], offer a suite of features, including both vertex-centric processing and queries, and, sensibly, may offer a graph processing system developed on top of a graph database. Still, graph databases and TLAV frameworks are decidedly different, and not fit for performance comparison.

This survey has largely ignored the vast body of research on vertex-centric, or distributed, algorithms, because, rather than focusing on the development of novel vertex-centric algorithms themselves, TLAV frameworks are more like an analytics operating system that facilitate the execution such algorithms on large datasets. While many vertex-centric, or distributed, algorithms have been long-established [Lynch 1996], the introduction of TLAV frameworks has also spurred further distributed algorithm development [Quick et al. 2012].

6. CONCLUSIONS

Think Like a Vertex frameworks are distributed, fault-tolerant systems that implement a programming model for iterative graph algorithms. TLAV frameworks are characterized by functions that are executed on individual graph vertices, and generally only interact with data from adjacent vertices along incident edges. This vertex-centric model allows for efficient decentralized computation.

Many design decisions can significantly impact the performance of TLAV systems. The timing of the system can be synchronous or asynchronous, vertices can communicate through message passing or shared memory, and the user-defined vertex function can be implemented at various levels of granularity. A recurring design trade-off is performance increase at the expense of added complexity and decreased scalability. Synchronous message-passing frameworks are simple to implement and scale linearly with graph size, but at the expense of a high fixed-overhead because of the synchronization barrier. In contrast, asynchronous shared-memory systems can significantly outperform synchronous frameworks, but don't scale as well. When implementing a TLAV system, one must consider the types of algorithms to be executed and the underlying graph when making design decisions.

TLAV frameworks introduce an interesting algorithmic perspective. Like distributed algorithms, vertex programs are employed at a local level in order to achieve a global result. Implemented algorithms typically require shorter runtime at the expense of increased communication between vertex program kernels. A trade-off is exposed where shared-memory algorithms are conceptually simple and perform better, while distributed algorithms are more constrained but offer greater scalability. Several alternative frameworks have been developed that combine local and global perspectives for improved performance relative to TLAV frameworks.

Three observations can be made regarding the direction of future work. First, there appears to be a lot of room for improvement regarding the analysis of dynamic networks, especially in the distributed TLAV domain. At the very least it must be asked, due to the offline nature of TLAV frameworks, is the vertex-centric model a viable approach for dynamic graph analysis? While frameworks such as Kineograph and Chronos demonstrate that such platforms can be developed, a large timegap remains between the current capabilities of temporal TLAV frameworks [Cheng et al. 2012; Han et al. 2014] and the timeliness requirements for real-time processing applications.

The second observation regards the development of hybrid programming models. The vertex programming model is interesting because it offers a new, local perspective that heavily contrasts with the global perspective of traditional algorithms. Several alternatives to the vertex-centric programming model were presented in Section 4.4, and examples abound where local-global hybrid models increased performance for a niche of algorithms. Inspired by [Beamer et al. 2013], further opportunities may exist in hybridizing graph processing perspectives.

Lastly, while TLAV frameworks have become established as a viable approach for large-scale graph processing, including industry adoption [Ching 2013], others have argued that the constraint of vertex programming isn't worth the scalability improvements, as nearly any graph can still fit in memory of a high-end server [Shun and Blelloch 2013]. This claim has been supported in practice [Gupta et al. 2013]. Nevertheless, graphs will continue to grow in size, but with the ingenuity behind

single-node platforms, advances in memory density, storage and GPU hardware, the necessity of distribution still remains an open question, and the trade-offs between computation expressivity and scalability warrant further investigation.

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, USA, 124–124. 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

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 of Computing Systems* 39, 6 (2006), 929–939.

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

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 Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '07)*. ACM, New York, NY, USA, 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 Twenty-Third International Joint Conference on Artificial Intelligence (IJCAI '13)*. AAAI Press, 461–467. http://dl.acm.org/citation.cfm?id=2540128.2540196

D. P. Bertsekas, F. Guerriero, and R. 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, Inc., Upper Saddle River, NJ, USA.

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, USA, 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. *The VLDB Journal* 21, 2 (April 2012), 169–190. DOI: http://dx.doi.org/10.1007/s00778-012-0269-7

Aydín Buluc̈c 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

Zhuhua Cai, Dionysios Logothetis, and Georgos Siganos. 2012. Facilitating Real-time Graph Mining. In *Proceedings of the Fourth International Workshop on Cloud Data Management (CloudDB '12)*. ACM, New York, NY, USA, 1–8. DOI: http://dx.doi.org/10.1145/2390021.2390023

K. M. Chandy and J. 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

Shuai Che. 2014. GasCL: A Vertex-Centric Graph Model for GPUs. *IEEE High Performance Extreme Computing Conference (HPEC)* (2014).

Qun Chen, Song Bai, Zhanhuai Li, Zhiying Gou, Bo Suo, and Wei Pan. 2014. 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. (2014).

Rong Chen, Jiaxin Shi, Yanzhe Chen, Haibing Guan, and Haibo Chen. 2013. *Powerlyra: Differentiated graph computation and partitioning on skewed graphs*. Technical Report. Technical Report IPADSTR-2013-001, Shanghai Jiao Tong Univ.

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 Third ACM Symposium on Cloud Computing (SoCC '12)*. ACM, New York, NY, USA, 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 Eleventh International Conference on Information and Knowledge Management (CIKM '02)*. ACM, New York, NY, USA, 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, USA, 85–98. DOI: http://dx.doi.org/10.1145/2168836.2168846

Avery Ching. 2013. Scaling Apache Giraph to a trillion edges. Retrieved December 19, 2014 from https://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges/10151617006153920. (2013).

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, USA, 302 – 308. DOI: http://dx.doi.org/10.1109/IPPS.1996.508073

Jonathan Cohen. 2009. Graph Twiddling in a MapReduce World. *Computing in Science and Engg.* 11, 4 (July 2009), 29–41. DOI: http://dx.doi.org/10.1109/MCSE.2009.120

Grzegorz Czajkowsi. 2009. Large-Scale graph computing at Google. Retrieved December 4, 2014 from http://googleresearch.blogspot.com/2009/06/large-scale-graph-computing-at-google.html. (2009).

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. of Global Optimization* 49, 3 (March 2011), 425–434. DOI: http://dx.doi.org/10.1007/s10898-010-9568-y

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

E. W. 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, USA, 810–818. DOI: http://dx.doi.org/10.1145/1851476.1851593

Arash Fard, Amir Abdolrashidi, Lakshmish Ramaswamy, and John A Miller. 2012. Towards efficient query processing on massive time-evolving graphs.. In *CollaborateCom*. 567–574.

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* (1977), 35–41.

Zhisong Fu, Michael Personick, and Bryan Thompson. 2014. MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs. In *Proceedings of Workshop on GRAph Data Management Experiences and Systems (GRADES'14)*. ACM, New York, NY, USA, Article 2, 6 pages. DOI: http://dx.doi.org/10.1145/2621934.2621936

J. Gehweiler and H. Meyerhenke. 2010. A distributed diffusive heuristic for clustering a virtual P2P supercomputer. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. 1–8. DOI: http://dx.doi.org/10.1109/IPDPSW.2010.5470922

Abdullah Gharaibeh, Elizeu Santos-Neto, Lauro Beltrão Costa, and Matei Ripeanu. 2013. The Energy Case for Graph Processing on Hybrid CPU and GPU Systems. , Article 2 (2013), 8 pages. DOI: http://dx.doi.org/10.1145/2535753.2535755

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, USA, 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, USA, 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 *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* (2014). 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* (2014). http://arxiv.org/abs/1410.4885

Philipp Haller and Heather Miller. 2011. Parallelizing machine learning-functionally: A framework and abstractions for parallel graph processing. In *2nd Annual Scala Workshop*.

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.

Wentao Han, 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 Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, Article 1, 14 pages. DOI: http://dx.doi.org/10.1145/2592798.2592799

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, USA, 77–85. DOI: http://dx.doi.org/10.1145/2487575.2487581

Li-Yung Ho, Tsung-Han Li, Jan-Jan Wu, and Pangfeng Liu. 2013. Kylin: An efficient and scalable graph data processing system. In *Proceedings of the 2013 IEEE International Conference on Big Data*. 193–198.

Petter Holme and Jari Saramäki. 2012. Temporal networks. *Physics reports* 519, 3 (2012), 97–125.

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, USA, 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*.

IBM, Paul Zikopoulos, and Chris Eaton. 2011. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data* (1st ed.). McGraw-Hill Osborne Media.

Borislav Iordanov. 2010. HyperGraphDB: A Generalized Graph Database. In *Proceedings of the 2010 International Conference on Web-age Information Management*. Springer-Verlag, Berlin, Heidelberg, 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 *First International Workshop on Graph Data Management Experiences and Systems (GRADES '13)*. ACM, New York, NY, USA, 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, USA, 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 Ninth IEEE International Conference on Data Mining (ICDM '09)*. IEEE Computer Society, Washington, DC, USA, 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, USA, 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 System Technical Journal* 49, 2 (1970), 291–307. DOI: http://dx.doi.org/10.1002/j.1538-7305.1970.tb01770.x

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, USA, 169–182. DOI: http://dx.doi.org/10.1145/2465351.2465369

Udayan Khurana. 2013. *An Introduction to Temporal Graph Data Management*. Technical Report. Technical report, University of Maryland College Park.

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

Vassilis Kostakos. 2009. Temporal graphs. *Physica A: Statistical Mechanics and its Applications* 388, 6 (2009), 1007–1023.

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, USA, 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* (2014). http://arxiv.org/abs/1403.0701

Alan G. Labouseur, Paul W. Olsen, Kyuseo Park, and Jeong-Hyon Hwang. 2014. A Demonstration of Query-oriented Distribution and Replication Techniques for Dynamic Graph Data. In *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web Companion (WWW Companion '14)*. International World Wide Web Conferences Steering Committee, Geneva, Switzerland, 127–130. DOI: http://dx.doi.org/10.1145/2567948.2577026

Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2007. Graph Evolution: Densification and Shrinking Diameters. *ACM Trans. Knowl. Discov. Data* 1, 1, Article 2 (March 2007). DOI: http://dx.doi.org/10.1145/1217299.1217301

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 Mathematics* 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 Eighth Workshop on Mining and Learning with Graphs (MLG '10)*. ACM, New York, NY, USA, 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* (2010).

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, USA, 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. *Proceedings of the VLDB Endowment* 8, 3 (2014).

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

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

Tiago Alves Macambira and Dorgival Guedes. 2010. A Middleware for Parallel Processing of Large Graphs. In *Proceedings of the 8th International Workshop on Middleware for Grids, Clouds and e-Science (MGC '10)*. ACM, New York, NY, USA, Article 7, 6 pages. DOI: http://dx.doi.org/10.1145/1890799.1890806

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, USA, 135–146. DOI: http://dx.doi.org/10.1145/1807167.1807184

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

Henning Meyerhenke, Peter Sanders, and Christian Schulz. 2014. Parallel Graph Partitioning for Complex Networks. *arXiv preprint arXiv:1404.4797* (2014).

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 Tenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '99)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 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 Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 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 2014)*. ACM, New York, NY, USA, Article 4, 12 pages. DOI: http://dx.doi.org/10.1145/2611354.2611365

Huazhong Ning, Wei Xu, Yun Chi, Yihong Gong, and Thomas S Huang. 2007. Incremental Spectral Clustering With Application to Monitoring of Evolving Blog Communities.. In *SDM*. SIAM, 261–272. DOI: http://dx.doi.org/10.1137/1.9781611972771.24

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, USA, 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

Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C. Hoe, Jos F. Martnez, and Carlos Guestrin. 2014. GraphGen: An FPGA Framework for Vertex-Centric Graph Computation. In *Proceedings of the 2014 IEEE 22Nd International Symposium on Field-Programmable Custom Computing Machines (FCCM '14)*. IEEE Computer Society, Washington, DC, USA, 25–28. DOI: http://dx.doi.org/10.1109/.13

Matthew Felice Pace. 2012. {BSP} vs MapReduce. *Procedia Computer Science* 9, 0 (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, USA, 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, USA.

Ivanilton Polato, Reginaldo R, Alfredo Goldman, and Fabio Kon. 2014. A comprehensive view of Hadoop research – A systematic literature review. *Journal of Network and Computer Applications* 46, 0 (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, USA, 4–4. 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, Heidelberg, 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* (1st ed.). IEEE Computer Society Press, Los Alamitos, CA, USA.

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 2012) (ASONAM '12)*. IEEE Computer Society, Washington, DC, USA, 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. *Physical Review E* 76, 3 (2007), 036106. http://dx.doi.org/10.1103/PhysRevE.76.036106

Fatemeh Rahimian, AmirH. 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 Heidelberg, 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, USA, 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, USA, 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 Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 472–488. DOI: http://dx.doi.org/10.1145/2517349.2522740

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)*. ACM, New York, NY, USA, 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. (2014), 979–990. DOI: http://dx.doi.org/10.1145/2588555.2610518

Kirk Schloegel, George Karypis, and Vipin Kumar. 2001. Wavefront Diffusion and LMSR: Algorithms for Dynamic Repartitioning of Adaptive Meshes. *IEEE Trans. Parallel Distrib. Syst.* 12, 5 (May 2001), 451–466. DOI: http://dx.doi.org/10.1109/71.926167

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 Second International Conference on Cloud Computing Technology and Science (CLOUDCOM '10)*. IEEE Computer Society, Washington, DC, USA, 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 *Problems of Cybernetics and Informatics (PCI), 2012 IV International Conference*. 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 2013) (ICDE '13)*. IEEE Computer Society, Washington, DC, USA, 553–564. DOI: http://dx.doi.org/10.1109/ICDE.2013.6544855

Bin Shao, Haixun Wang, and Yatao Li. 2013a. 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, USA, 505–516. DOI: http://dx.doi.org/10.1145/2463676.2467799

Yingxia Shao, Junjie Yao, Bin Cui, and Lin Ma. 2013b. PAGE: A Partition Aware Graph Computation Engine. In *Proceedings of the 22Nd ACM International Conference on Conference on Information & Knowledge Management (CIKM '13)*. ACM, New York, NY, USA, 823–828. DOI: http://dx.doi.org/10.1145/2505515.2505617

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, USA, 135–146. DOI: http://dx.doi.org/10.1145/2442516.2442530

Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, and Viktor Prasanna. 2014a. 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

Yogesh Simmhan, Charith Wickramaarachchi, Alok Kumbhare, Marc Frincu, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, and Viktor Prasanna. 2014b. Scalable Analytics over Distributed Time-series Graphs using GoFFish. *arXiv preprint arXiv:1406.5975* (2014). http://arxiv.org/abs/1406.5975

George M. Slota, Kamesh Madduri, and Sivasankaran Rajamanickam. 2014. PULP: Scalable Multi-Objective Multi-Constraint Partitioning for Small-World Networks. (2014).

Isabelle Stanton. 2014. Streaming Balanced Graph Partitioning Algorithms for Random Graphs. In *Proceedings of the Twenty-Fifth 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, USA, 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 Part I*. Springer-Verlag, Berlin, Heidelberg, 764–780. http://dl.acm.org/citation.cfm?id=1940281.1940330

Jukka Suomela. 2013. Survey of Local Algorithms. *ACM Comput. Surv.* 45, 2, Article 24 (March 2013), 40 pages. DOI: http://dx.doi.org/10.1145/2431211.2431223

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, USA, 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, Heidelberg, 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". *Proceedings of the VLDB Endowment* 7, 3 (2013).

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, USA, 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, USA, 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 Sixth ACM International Conference on Web Search and Data Mining (WSDM '13)*. ACM, New York, NY, USA, 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* (2013). 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* (2014). 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, USA, 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, USA, 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 *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*. 568–579. DOI: http://dx.doi.org/10.1109/ICDE.2014.6816682

Rui Wang and K. Chiu. 2013. A stream partitioning approach to processing large scale distributed graph datasets. In *Big Data, 2013 IEEE International Conference on*. 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, USA, 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, USA, 235–244. DOI: http://dx.doi.org/10.1145/1995896.1995934

C Xie, R Chen, H Guan, B Zang, and H 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, Inc., 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. *Proceedings of the VLDB Endowment* 7, 14 (2014).

Jilong Xue, Zhi Yang, Zhi Qu, Shian Hou, and Yafei Dai. 2014. Seraph: An Efficient, Low-cost System for Concurrent Graph Processing. (2014), 227–238. DOI: http://dx.doi.org/10.1145/2600212.2600222

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

Da Yan, James Cheng, Kai Xing, Li Lu, Wilfred Ng, and Yingyi Bu. 2014b. Pregel Algorithms for Graph Connectivity Problems with Performance Guarantees. In *Proceedings of the VLDB Endowment*, Vol. 7.

Shengqi Yang, Xifeng Yan, Bo Zong, and Arijit Khan. 2012. Towards Effective Partition Management for Large Graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 517–528. DOI: http://dx.doi.org/10.1145/2213836.2213895

Eiko Yoneki and Amitabha Roy. 2013. Scale-up Graph Processing: A Storage-centric View. In *First International Workshop on Graph Data Management Experiences and Systems (GRADES '13)*. ACM, New York, NY, USA, 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. (2014), 401–412. DOI: http://dx.doi.org/10.1109/SC.2014.38

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2228298.2228301

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, USA, 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, USA, 61–68. DOI: http://dx.doi.org/10.1145/2351316.2351325

Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. 2012a. Accelerate Large-scale Iterative Computation Through Asynchronous Accumulative Updates. (2012), 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, USA, 717–724. DOI: http://dx.doi.org/10.1109/BigData.Congress.2014.106

Da Zheng, Randal Burns, and Alexander S. Szalay. 2013. Toward Millions of File System IOPS on Low-cost, Commodity Hardware. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, Article 69, 12 pages. DOI: http://dx.doi.org/10.1145/2503210.2503225

Jianlong Zhong and Bingsheng He. 2014. Medusa: Simplified Graph Processing on GPUs. *Parallel and Distributed Systems, IEEE Transactions on* 25, 6 (June 2014), 1543–1552. DOI: http://dx.doi.org/10.1109/TPDS.2013.111

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, Heidelberg, 337–348. DOI: http://dx.doi.org/10.1007/978-3-540-68880-8_32